USING AUTOMATED VIDEO PROCESSING TO IDENTIFY PEDESTRIAN-VEHICLE CONFLICTS

A Dissertation
Presented to
The Academic Faculty

by

Spencer Maddox

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Civil and Environmental Engineering

Georgia Institute of Technology
May 2020

USING AUTOMATED VIDEO PROCESSING TO IDENTIFY PEDESTRIAN-VEHICLE CONFLICTS

Approved by:

Dr. Kari Watkins, Advisor
School of Civil and Environmental Engineering
Georgia Institute of Technology

Colin Usher
Senior Research Scientist
Georgia Tech Research Institute

Dr. Randall Guensler
School of Civil and Environmental Engineering
Georgia Institute of Technology

Date Approved: April 24, 2020

To pedestrians everywhere

ACKNOWLEDGEMENTS

As I reflect on this thesis and my pending graduation, the path I took feels like one that was affected by a butterfly's wings. As an undergraduate, I was exposed to the Global Engineering Leadership minor, and without it, I would have never have met my advisor, Dr. Watkins. I would like to thank Dr. Kennedy for her work in developing the minor which impacted me so greatly.

Dr. Watkins, I would like to thank you for transforming my academic and professional career. Without Sustainable Transportation Abroad, I likely would have never found my passion with pedestrians, cyclists, and transit. I do not where I would be currently without that passion.

I would like to thank the Georgia Department of Transportation for their commitment to improving pedestrian safety. Without them, this project and thesis would have never been possible.

Colin, I would like to thank you for allowing me to work on this project. Under your wing, I learned more about computer vision, neural networks, and coding than I ever thought I would have. This project challenged me and put me out of my comfort zone, and with that I learned many new skills along the way. I would also like to thank the project team; Aravind and Nadun, I would have been lost without working beside you, and I would like to thank you for your patience with me.

Lastly, I would like to thank my family. To my mother and father, I would not be here without you both, and I would like to thank you for all your guidance, love, and support. Garrett, I would like to thank you for adopting your dog. He gave me the joy and

mental breaks I needed to complete this thesis. Jennifer, my fiancée, I would like to thank you for all the walks we took in Atlanta, and for you listening to my rants about urban infrastructure and pedestrian safety. You kept motivated under the unusual circumstances which I completed this thesis – the coronavirus. I cannot wait until we are pedestrians once again exploring our urban environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

CSV	Comma separated values
DST	Deceleration to safety time
GDOT	Georgia Department of Transportation
GT	Gap Time
GTr	Ground Truth
GTRI	Georgia Tech Research Institute
PET	Post-encroachment Time
SQL	Structured Query Language
TA	Time to Accident
TCT	Traffic Conflicts Technique
TTC	Time to Collision
TTV	Time to Vehicle

SUMMARY

From 2008 to 2017, pedestrian fatalities in the United States increased from 4,414 to 5,977 (“Traffic Safety Facts”). While crashes have increased across the entire country, on an individual roadway, pedestrian fatalities remain a rare occurrence. Even if a corridor has multiple fatalities, it is difficult to identify whether those crashes are statistically significant or random occurrences (Svensson and Hydén 2006). Therefore, crash data alone does not provide adequate information on where state transportation agencies should place mitigation efforts, such as pedestrian hybrid beacons and traffic calming.

Surrogate safety measures, such as near misses, provide transportation agencies with additional data on specific corridors. Historically, near miss analysis has been performed manually, either by watching recorded video or performing it on site. These studies are both time consuming and intensive. With the advances in video processing and object detection, automated processing software provides a less-intensive method to identify pedestrian-vehicle conflicts. Previous research with automated video processing still requires an annotated scene to identify and categorize conflicts (Saunier et al 2007).

The Georgia Tech Research Institute (GTRI) developed a suite of software tools for Georgia Department of Transportation (GDOT). These tools can identify pedestrians, vehicles, and cyclists, filter out false positives, and playback the detection in the video. Using GTRI’s existing software, this thesis develops a practical conflict identification tool without intensive calibration.

This conflict detection tool allows GDOT to understand conflict severity and location. Additionally, for verification purposes conflicts can be watched when they occur within the video. This conflict analysis software provides GDOT an important surrogate safety measure to improve pedestrian safety. The conflict analysis software, developed by this thesis research, accurately identifies conflicts and calculates the pedestrian-vehicle post-encroachment times.

CHAPTER 1. INTRODUCTION

As pedestrian fatalities have risen in the United States within the last 10 years, they have composed an increasing share of overall transportation fatalities. Out of all pedestrian fatalities in the United States, 80% occur in urban areas and 72% occur at non-intersection locations ("Traffic Safety Facts"). While overall numbers have increased, crashes between pedestrians and vehicles on an individual corridor or location remain rare occurrences. From crash data alone, a traffic engineer cannot differentiate between a statistically significant crash location versus a random occurrence (Svensson and Hydén 2006).

Additionally, understanding an individual pedestrian's risk in a location requires an exposure rate. To calculate a pedestrian's exposure, accurate pedestrian counts and crossing locations are required. Traditionally, pedestrian counts have been completed manually. Even with a manual count, however, the location where pedestrians crossed is not often logged. Without crossing location, identifying where to place intervention measures such as pedestrian hybrid beacons is difficult. Traffic engineers, therefore, cannot identify dangerous corridors which require intervention solely from crash statistics.

Specifically in Metro Atlanta, multiple corridors have a high frequency of non-intersection crossing locations. These non-intersection crossing locations, such as mid-block crossings can be higher risk than other roadway elements ("Traffic Safety Facts"). Low intersection density can cause mid-block crossings (Cherry et al 2012). When considering intervention measures, Georgia Department of Transportation (GDOT) must know the number of pedestrians, crossing locations, and the way pedestrians interact with vehicles.

To improve pedestrian safety, GDOT tasked the Georgia Tech Research Institute (GTRI) to develop an automated pedestrian tracker. This initial project included four key pieces: a portable trailer, tracking software, report generation, and playback tool. Together, these tools provide a user friendly pedestrian tracking system. The solar powered portable trailer system has four high definition cameras and can record data for multiple days.

As an extension of this project, GTRI upgraded the pedestrian tracking software to track vehicles and cyclists as well. GDOT also tasked GTRI with exploring a pedestrian-vehicle conflict metric. While the pedestrian counts and crossing locations provide meaningful data, these metrics alone fail to determine the safety of those crossings. By developing a pedestrian-vehicle conflict metric, unsafe pedestrian-vehicle interactions are identified. Figure 1 below demonstrates the overall workflow and describes each step.

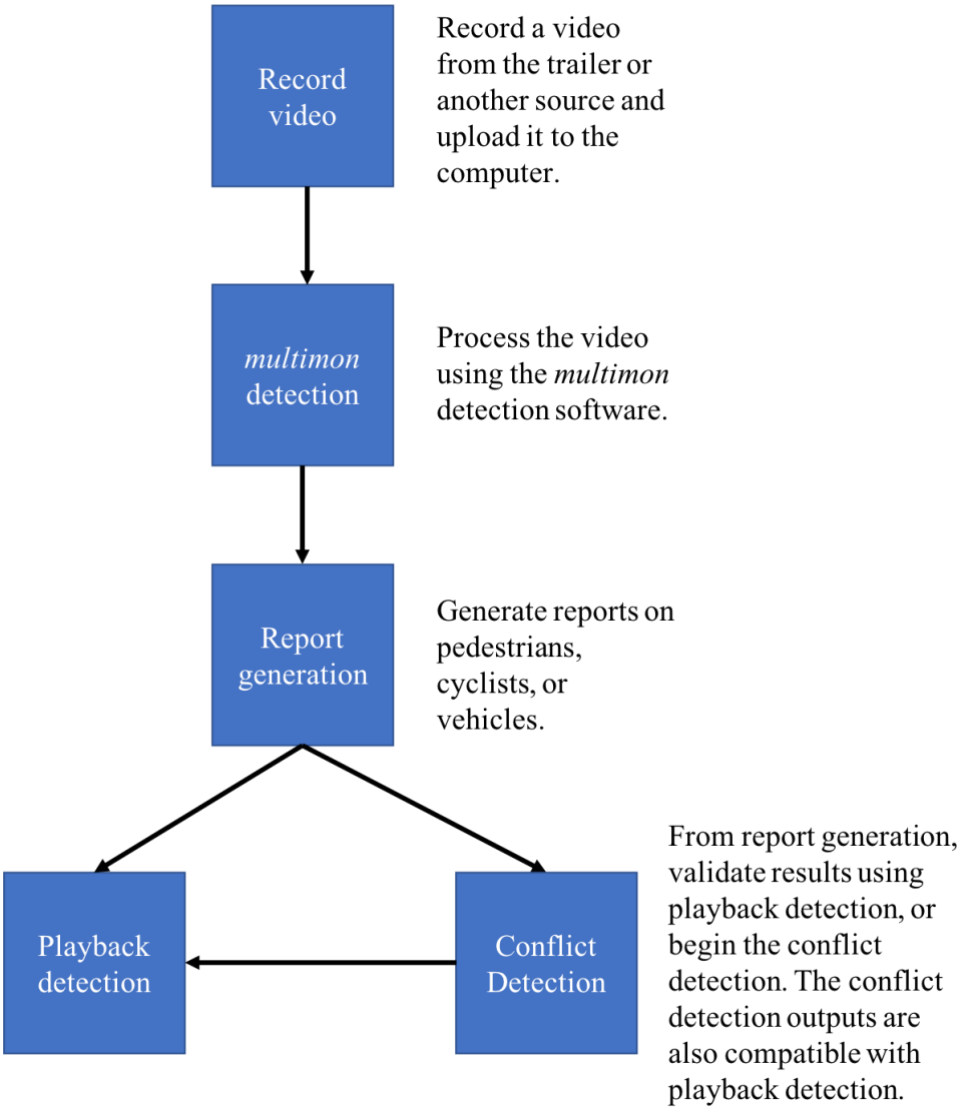


Figure 1 – Automated Tracker Tools and Workflow

The Multimon tracker identifies pedestrians, vehicles, and cyclists from a video feed, and saves their coordinates in an SQL database. Combined with Report Generation, this system provides both the respective pedestrian, vehicle, or cyclist count, and visualizes their respective trajectories. After report generation, playback detection allows users to verify pedestrian, vehicle, cyclist, and conflict detection. The conflict detection tool must run after report generation, and this tool identifies and displays pedestrian-vehicle conflicts.

CHAPTER 2. LITERATURE REVIEW

2.1 Overview

This literature review will discuss the latest research in automated video processing, pedestrian crossing behavior, and surrogate conflict analysis.

2.2 Pedestrian Crossing Behavior

Depending on the environment's infrastructure, signalized crossings may exist at low densities. Therefore, pedestrians do not always cross at an intersection. When deciding whether to walk to the nearest intersection or cross unsignalized, a pedestrian calculates his or her associated risk (Schneider et al 2004). In China, one study showed that the most significant illegal crossings occur when the legal walking distance was five times the illegal one (Cherry et al 2012). In Las Vegas, mid-block crossings spike 150 feet from the nearest intersection (Cui and Nambison 2003). As distances increase between legal crossings, pedestrians perceive the benefit of saved time by crossing illegally to outweigh the risk of crossing illegally.

Pedestrians target a level of risk that maximizes the difference between perceived benefits and cost of a choice (Schneider et al 2004). Differences, however, exist between perceived safety and number of crashes (Schneider et al 2004). This risk calculation is not the same for each pedestrian. Jiang, Wang, Bengler, and Guo compared gap acceptance between pedestrians in China and Germany. The results of the study found that pedestrians in Germany will accept a rolling gap, where they begin to cross after the first car gap, while pedestrians in China will wait for an entire platoon to end before crossing (Jiang et al 2015).

A different pedestrian behavior study in China found that pedestrians will accept a rolling gap rather than wait for all lanes to clear (Cherry et al 2012). These results suggests that researchers remain unsure about pedestrian behavior at unsignalized crossings.

Pedestrian behavior can differ depending on the infrastructure as well, especially as applied to pedestrian speeds and delay. Ishaque and Noland found that pedestrians speeds are slowest at Zebra crossings, followed by pedestrian refuges and pelican crossings, and are highest at random, unmarked crossings (Ishaque and Noland 2008). These results suggest that pedestrian speed has a positive relationship with perceived risk. In addition to identifying changes in pedestrian behavior at different infrastructure, the risk between each pedestrian-vehicle conflict changes as well.

2.3 Surrogate Conflict Measures for Pedestrian-Vehicle Interactions

In 1980, Cynecki defined different conflict types and the likely resulting severity. Cynecki defines 13 different types of conflict, 7 types of pedestrian conflict severity, and 6 types of vehicle conflict severity. To properly categorize pedestrian-vehicle conflicts, manual observers were used. While the large number of categorizations provide detailed context for conflicts, conducting analysis requires training and is both time and labor intensive.

Using the Swedish Traffic Conflicts Technique (TCT) method, which includes time-to-accident (TA) and post-encroachment time (PET), Svensson and Hydén define a hierarchy for pedestrian-vehicle conflicts. From this methodology, the highest severity conflicts are infrequent; highest severity conflict are considered crashes and situations which no one puts themselves in deliberately (Svennson and Hydén 2006). Comparatively,

conflicts at fairly high severities occur much more frequently. Fairly high severities are more likely to occur at a non-signalized intersection. Svensson and Hydén hypothesize that the frequency of fairly high severity conflict reduces the number of crashes since road users expect these conflicts to occur. Conversely, signalized intersections have a low frequency of relatively high severity conflicts, but crashes still occur (Svensson and Hydén 2006). From this research, the relationship between conflicts and crashes may not be linear.

TA, also called time-to-collision (TTC), is defined as "the time that remains until a collision between two objects would have occurred if the collision course and speed difference are maintained". A major limitation of TTC is the extrapolation required from field work (Ismail et al 2009). While limitations of TTC exist, many studies use TTC as the surrogate safety metric. A difference exists in a vehicle's TTC and a pedestrian's time to vehicle (TTV) which is seen in Figure 2 (Matsui et al 2013).

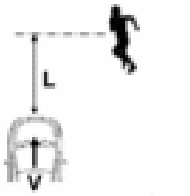
Focused object	Vehicle	Pedestrian
Time	Vehicle time to collision (Vehicle TTC)	Pedestrian time to vehicle (Pedestrian TTV)
Definition	 Vehicle TTC = $\frac{L}{V}$	 Pedestrian TTV = $\frac{Ld}{v}$
Study	Previous study (Matsui et al. 2011b)	Present study

Figure 2 - Vehicle TTC and Pedestrian TTV Definition in Matsui et al 2013

Using videos from cameras installed on taxis in Japan, Matsui et al showed that the TTC depends on the obstruction amount of the pedestrian. The shortest TTC occurred when a pedestrian moved out from vehicles in the other lane. Additionally, TTV was lower in situations without crosswalks than those with crosswalks (Matsui et al 2013). This study demonstrates that the amount of obstruction and presence of infrastructure influences pedestrian-vehicle interactions.

TTC requires trajectory data for the vehicle. Trajectory data, however, is not always available. On a before and after study of an intersection in Atlanta, trajectory data could not be calculated for pedestrian-vehicle conflicts. Instead, manual observations were used to determine the number of near misses. The results of the study found an average of one near miss per every two hours, however, the near miss categorizations were "subject to the observer's perception of risk" (Watkins et al 2016).

Because TTC requires additional data, additional surrogate conflict measures have been identified in the data. Additional surrogate safety metrics include gap time and deceleration-to-safety time. Gap time (GT) is a variation of the PET that is calculated at each instant by projecting the movement of the interacting road users in space and time. Deceleration-to-safety time (DST) is the necessary deceleration to reach a nonnegative PET value if the conflicting road users remain unchanged (Ismail et al 2009).

DST and TTC have underlying assumptions with speed that cause inaccuracy and unreliability in the data. PET was found to be the most reliable measure (Ismail et al 2009). Ismail et al also found that PET is the most promising indicator for safety implementation, but it fails to accurately represent a close call with a car having come to a complete stop.

PET, however, can be combined with trajectory data to provide a stronger surrogate safety measure. This process can be done using automated video processing.

These studies discuss conflicts, however, have different definitions of what is a conflict. Svennson and Hyden use a continuous hierarchy to define conflict severity. As TA increases, the interaction severity decreases, but the vehicle speed plays a role as well. For example, at high speeds a TA of 10 seconds can be considered a slight conflict which at lower speeds it would not be a conflict (Svennson and Hyden 2006). When simulating conflicts in Vissim, Wu et al found that a maximum PET of 8 seconds generated the best results (Wu et al 2016). Currently, there is no hard definition of a pedestrian-vehicle conflict in the literature, but conflicts with lower PET or TTC are considered more severe.

2.4 Automated Video Processing

Automated video processing is the process where software identifies different objects by analyzing videos frame by frame. Different types of trackers and tracking algorithms exist such as Kalman filtering and the Kanade–Lucas–Tomasi Feature algorithm (Sayed et al 2013). These algorithms use neural networks to identify different objects.

Previous studies have used video processing to identify and categorize pedestrian – vehicle interactions. Ismail, Sayed, Saunier, Autey, and St-Aubin have conducted multiple studies on video processing and surrogate safety measures. These studies have been conducted at case study intersections, and include manual annotation of the video scene. By manually annotating the video scene, real word coordinates can be converted from pixel coordinates. Different methodologies of annotating include using vehicles wheel base, identifying a series of points, or using a reference grid (Ismail et al 2009, Ismail et al 2013,

Saunier et al 2010). Additionally, using image projection and high mounted cameras, 3D videos can be projected into 2D (Laureshyn and Ardö 2006). With manual annotation or video projection, data such as object speed and acceleration are available.

The ability to calculate an individual object speeds provides improvements to surrogate safety measures. With speed, the time to collision can be calculated. With frame by frame speed, probabilistic models can be applied for collisions looking at frame by frame trajectories (Saunier et al 2008). This methodology has previously been applied for both vehicle-vehicle conflicts and pedestrian-vehicle conflicts. While annotation and calibration can provide improved surrogate safety measures, some drawbacks exist.

Annotation, calibration, and projection are only effective at flat intersections. At longer views with changing geography, the calibration becomes less accurate (Usher and Daley 2020). Even at flat intersections, parallax error, pixel resolution, and tracking errors can still occur (St-Aubin et al 2015). To avoid calibration errors from video processing, an alternate method to retrieve speed data is to use pneumatic loops.

By focusing the video at a crosswalk, the video processing can identify vehicles and pedestrians. To calculate speed and acceleration of vehicles, pneumatic loops were set up in multiple locations. Malkhamah et al show that deceleration rates of vehicles are a surrogate safety metric. Vehicle deceleration rates of 6 m/s2 were deemed a serious conflict, and deceleration rates of 4.5 m/s2 and 3.0 m/s2 were deemed slight conflicts and potential conflicts. With these three deceleration rates, the time to accident is 1.6, 1.8, and 2 seconds, respectively (Malkhamah et al 2005).

While previous studies exist in the literature that identify pedestrian-vehicle conflicts, each study still requires user input whether that be annotating a video, setting up calibration, or setting up pneumatic tubes. Additionally, each study was calibrated to an individual case location. In the literature, no software currently exists to be applied to a widespread location. The suite of tools developed by GTRI has flexibility to be used an applied to multiple locations. While the other methods provide speed data, they require site specific calibration. The benefit of the suite of software developed by GTRI and this thesis allow transportation engineers to identify conflicts without intensively calibrating the scene. Therefore, this video processing and conflict detection is more "automatic" than previous studies. This software can also be applied to non-intersection locations, where more than 72% of pedestrian fatalities occur ("Traffic Safety Facts").

2.5 Literature Review Summary

The existing literature demonstrates the validity of surrogate conflict analysis. While a combination of metrics may provide the most in-depth analysis of a conflict, PET is nonetheless one of the more robust metrics. The increase in fatalities in the United States demonstrates the need for mitigation strategies. Pedestrian behavior is unlikely to change; in areas with low intersection densities, pedestrians will likely choose to cross at an unmarked location. While previous studies have identified conflicts with speed data, these studies are limited to case study intersections. This thesis provides a conflict identification software with widespread, flexible application.

CHAPTER 3. GTRI'S AUTOMATED VIDEO PROCESSING TOOLS

3.1 Overview

This section describes GTRI's suite of tools for detecting, tracking, and counting pedestrians, vehicles, and cyclists developed for GDOT. GTRI developed three programs prior to this project for GDOT traffic engineers to use. These three software components include a tracker, report generator, and playback tool. Figure 1 depicts the way the tools interact with each other. This thesis work solely focuses on the development of the conflict detection software. The other associated tools were developed previously by project team members at GTRI. This section describes each software component and importance in conflict analysis.

3.2 Multimon: Pedestrian, Vehicle, and Bicycle Tracking

The Multimon software allows users to process videos and identify the number of pedestrians, vehicles, and cyclists as well as their frame by frame coordinates. This powerful tracker tool includes a user friendly interface to process videos. The software can process multiple camera feeds directly from the developed trailer, other streams, or even a single video. This flexibility allows multiple different types of datasets to be analyzed.

Once processing, the tracker differentiates and identifies pedestrians, vehicles, and bicycles. For each object, the Multimon software draws a bounding box around the object with its object id, frame, and (x,y) coordinates in pixel space. These data are written and stored in an SQL database. Figure 3 below shows a sample image of the tracker processing.

The overall detection accuracy for pedestrians, vehicles, and cyclists is 96%, 97%, and 91%, respectively (Usher and Daley 2020).

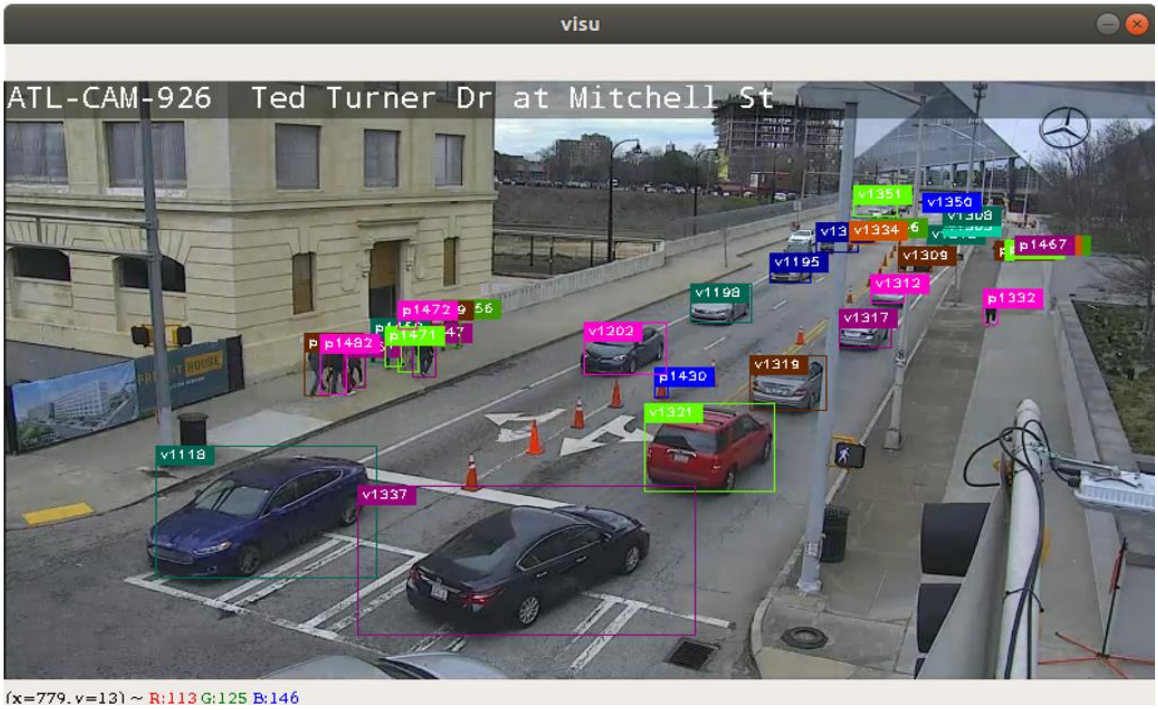


Figure 3 - Sample Tracker Visualization

3.3 Reportgen: Report Generation Tool

Following the processed videos, Reportgen is used to query the SQL database, filter out false positives, and generate images. The Reportgen software is also built with a user friendly interface similar to Multimon. At times, the tracker identifies false positives such as traffic signals, headlights, and fire hydrants as pedestrians. Additionally, pedestrians may be double counted due to occlusion.

For example, a pedestrian may walk behind a vehicle, and the tracker may give that pedestrian a new id. The Reportgen software filters out false positives and combines double-counted pedestrians. Ultimately, it provides a cleaner dataset to analyze conflicts

than the raw data from the tracker. In addition to a cleaner dataset, reportgen also outputs an overlay of each objects trajectory for the videos analyzed and the individual object trajectory in static images. The trajectory overlay image, as seen in Figure 4 in section 4.2 shows the density for the respective vehicle and pedestrian tracks.

3.4 Playbackdet: Playback Detection Tool

Playbackdet (playback detection tool) allows users to jump to individual detections using the text file generated from running Reportgen. Users can jump to, in video, where each individual object is detected. Rather than be required to watch videos constantly to identify pedestrians, the playback detection tool provides traffic engineers the ability to jump through pedestrian detections identified by the system rapidly.

3.5 Summary

Understanding the way each tool works provides the foundation for the conflict detection software. This thesis could not solely be a standalone tool, but was seamlessly integrated with the other tools – Multimon, Reportgen, and Playbackdet.

CHAPTER 4. CONFLICT IDENTIFICATION AND PET CALCULATION METHODOLOGY

4.1 Overview

This section details the methodology for identifying a pedestrian-vehicle conflict and calculating each conflict's post-encroachment time (PET). The conflict analysis relies on Reportgen's output. The outputs from Reportgen include all frame by frame (x,y) coordinates and time associated with each unique pedestrian and vehicle. Ultimately, the outputs from Reportgen are a list of points with specific IDs. To identify conflicts, the data must be filtered into manageable sections. After filtering the data, conflict identification logic is applied, then PET is calculated.

4.2 Report Outputs

Figure 3 below depicts sample visualizations from the report generation tool. On the left are the vehicle trajectories for one hour worth of data, and on the right are pedestrian trajectories for the same hour. The visualization shows that the individual points can be converted to lines, however, the conversion to lines would lose each points' associated time stamp.

Figure 4 - Sample Report Generation Outputs - Vehicle Tracks on the left with Pedestrian Tracks on the right

Additionally, intersecting each pedestrian trajectory line with each vehicle trajectory line would create thousands of potential "conflicts" for each pedestrian. Identifying these "conflicts" would not only create large computational constraints, but also fails to identify whether these "conflicts" occur within a meaningful timeframe. Intersecting all pedestrian and vehicle lines would therefore create additional steps to parse out actual conflicts. Therefore, a spacial-temporal point filtering methodology is applied.

4.3 Time Based Point Filtering

Each unique pedestrian and vehicle is a list of (x,y) coordinates with an associated time stamp. All conflict detection is processed over a 10 second moving time window. Similar to a moving average, this 10 second window filters the entire pedestrian and vehicle dataset in increments of 1 second in order to identify all intersecting trajectories within the window. Figure 5 below shows conceptually how this filtering occurs.

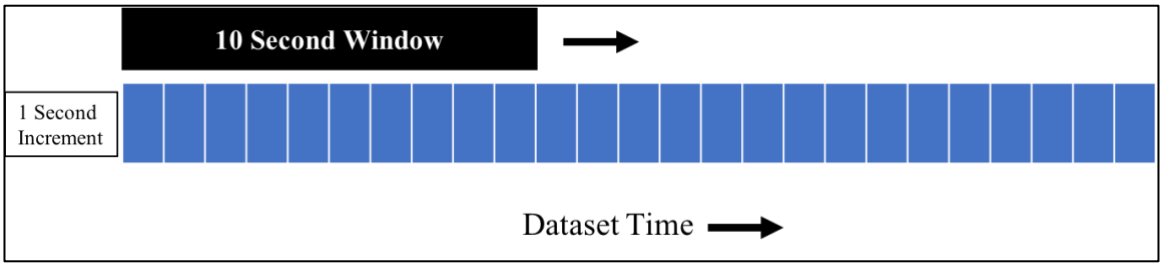


Figure 5 - Time Stepping through Dataset

The 10 second window will continue to increment by 1 second until it has stepped through the entire dataset. This time filtering provides a manageable slice of data to identify conflicts. While a 10 second PET is at the upper limit of Svensonn and Hydén's conflict

hierarchy, and 2 seconds greater than the maximum PET for Wu et al, his 10 second window ensures all conflicts are identified (Svensonn and Hydén 2006, Wu et al 2016). Additionally, the conflict software is flexible, and allows traffic engineers to choose a targeted conflict window which is described in APPENDIX A. For each 10 second window, the code looks for both horizontal and vertical conflicts for each unique pedestrian.

4.4 Conflict Identification

A conflict between a pedestrian and vehicle occurs in the horizontal direction when a unique pedestrian crosses horizontally and a vehicle conflicts with their trajectory vertically within the allotted time window. At this point in the conflict software, pedestrians and vehicles are still represented by a series of points. To identify a conflict, logic is applied. For a horizontal conflict, the logic is seen Figure 6 below. The blue dots represent a sample pedestrian's coordinates, and the orange dots represent a sample vehicle's coordinates for a given time, 10 seconds in this case.

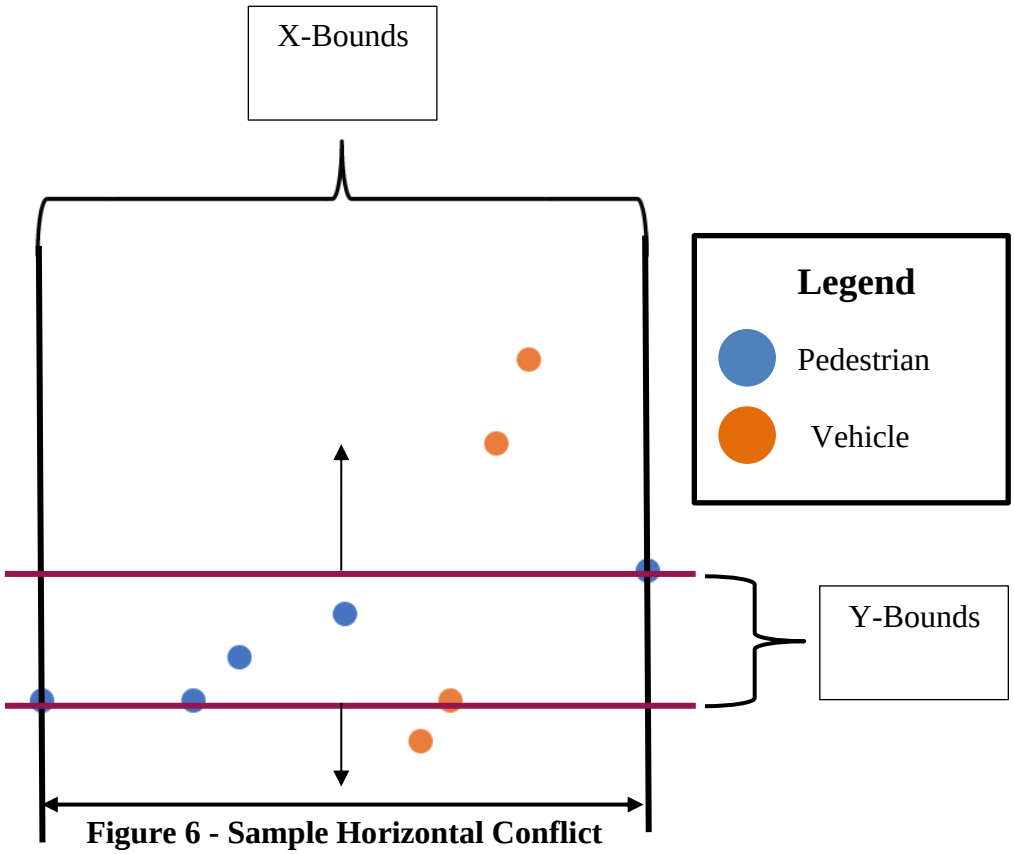


Figure 6 - Sample Horizontal Conflict

A conflict is assumed to occur when a unique vehicle has at least two points within a pedestrian's x-bounds which are represented by the black vertical lines on Figure 6. The same unique vehicle must also have at least one point outside the y-bounds which are represented by the horizontal red lines. If a unique vehicle's coordinates meet both these criteria, then a conflict is identified. Once a conflict is identified, it is stored in memory within the conflict detection program.

Like with horizontal conflicts, another check exists to identify vertical conflicts. Vertical conflicts occur when a pedestrian walks vertically and a vehicle crosses its path in the horizontal direction. Figure 7 shows a sample vertical conflict and the logic applied to identify it.

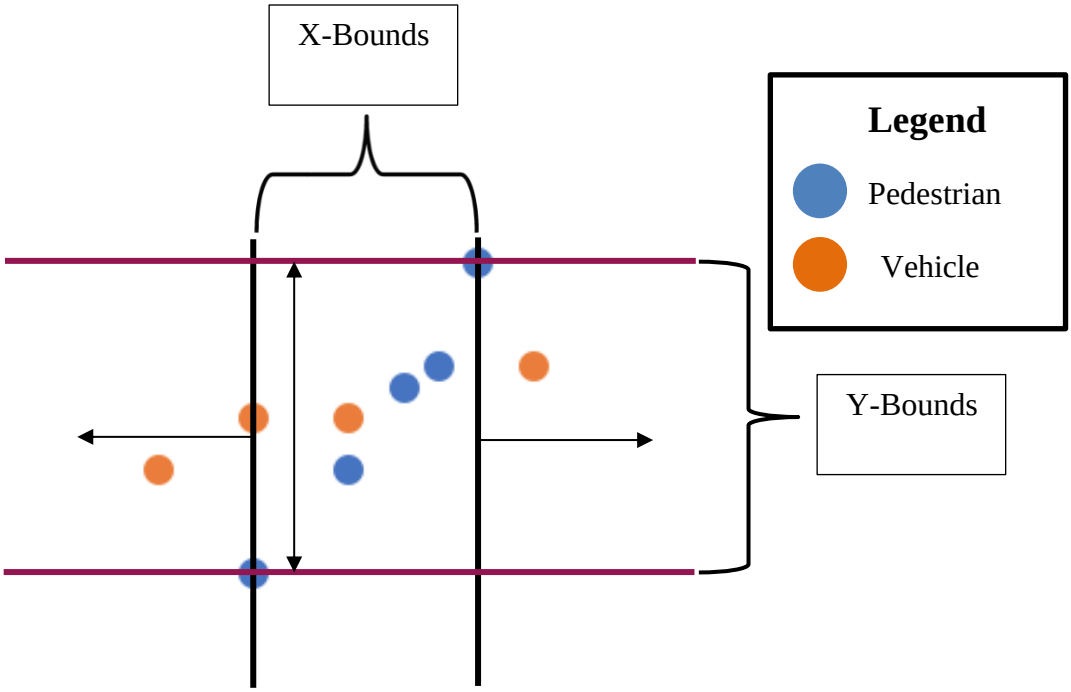


Figure 7 - Sample Vertical Conflict

A vertical conflict is assumed to occur when a unique vehicle has two points within a unique pedestrian's y-bounds and has two trajectory points outside the pedestrian's x-bounds. For vertical conflicts, the vehicle track is outside the x-bounds, and within the y-bounds bounds of the pedestrian track.

Through the application of this logic, both horizontal and vertical conflicts are identified. Additionally, the time window of 10 seconds is large enough to ensure this methodology successfully identifies all conflicts. For example, if a pedestrian or vehicle conflicts diagonally, the 10 second window is large enough to categorize this conflict as either horizontal or vertical. For each conflict identified, the software stores the time window of the conflict, and unique vehicle and pedestrian IDs in memory. This conflict identification and its storing allows for the software to perform additional conflict analysis.

4.5 Intersection Identification and PET Calculation

Conflict identification is the first step in the software's process. After processing the entire dataset, all identified conflicts are stored in the computer's memory. Identifying conflicts is not computationally insignificant. For a dataset of about 10 days, the conflict identification portion takes about 18 hours on an Intel I5 desktop system. From all identified conflicts, the software then finds the exact intersection point and calculates the PET. The methodology used ensures accurate intersection points and PET calculations.

4.6 Intersection Point

Once all conflicts are identified using the above assumptions, the exact point where the unique pedestrian and vehicle conflict must be found. Conceptually, the intersection point is easy to identify. Figure 8 below shows a conceptual example of an intersection point and PET calculation.

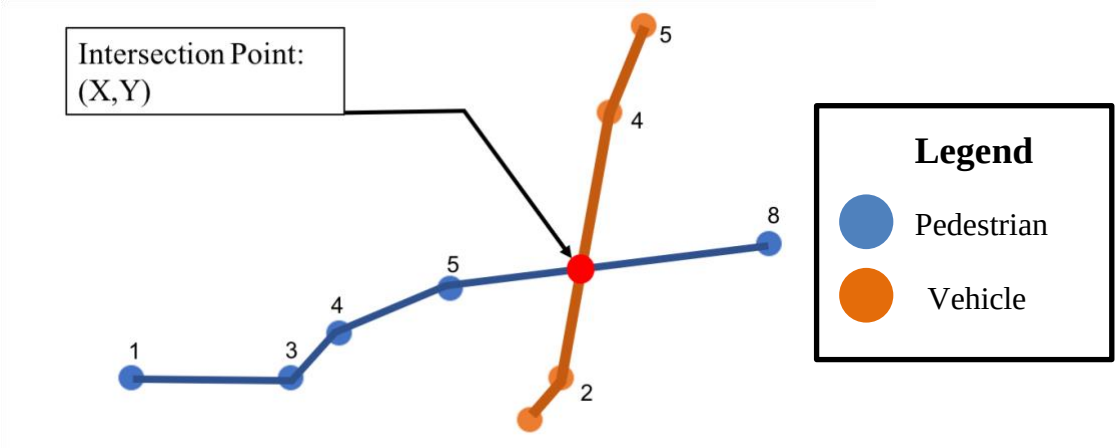


Figure 8 - Conceptual Intersection Point

The numbers on each point represent a point in time. Visually, one can see that the intersection point is caused by points with time 5 and 8 on the pedestrian line and points

with time 2 and 4 on the vehicle line. In Figure 8, the points which cause the intersection and associated time stamps are available. In Python, a software package exists to find the intersection point of two lines, however, the output only includes the intersection point, but not the points which cause it. Figure 9 below shows this initial iteration.

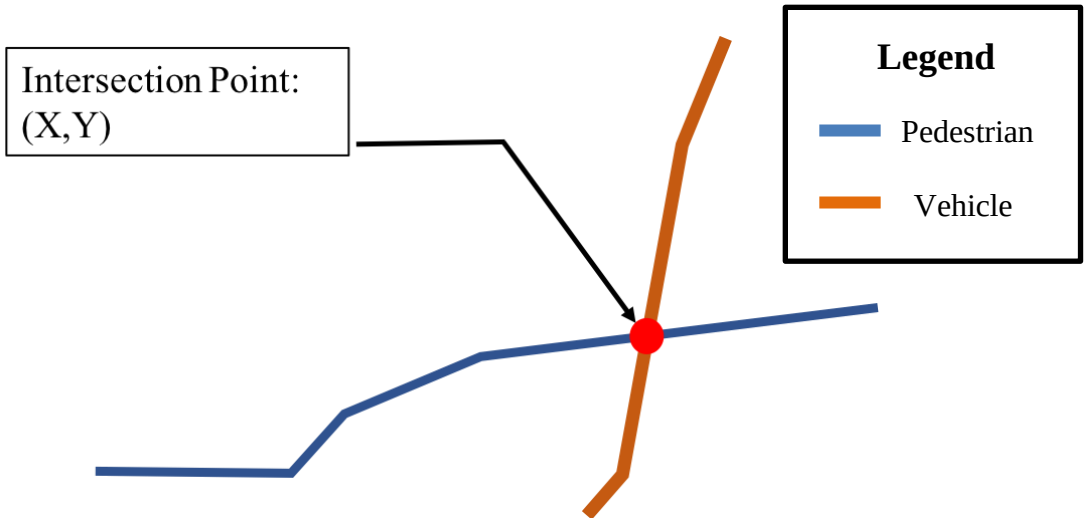


Figure 9 - Converting Points to Lines and Intersecting in Python

Figure 9 demonstrates that the conversion from points to lines only outputs the coordinates of the intersection point. This conversion eliminates individual points and their respective timestamp. Therefore, with this methodology, the points which cause the intersection cannot be determined. Ultimately, for PET calculations, the intersection point in addition to the points which cause the intersection are required. In Figure 8, the required points are those with time 5 and 8 for the pedestrian and those with time 2 and 4 for the vehicle.

The software initially checks to see if any pedestrian point is exactly the same with a vehicle point. If the coordinates are exactly the same, then that point is the intersection point. For a vast majority of conflicts, however, there is no exact overlap. To accurately

identify the intersection point and calculate the PET, a piecewise intersection was completed. By using multiple nested loops, the intersection point and points which cause the intersection can be found. The software steps through each conflict. For each conflict, each pedestrian segment is compared to each vehicle segment. Using this methodology, the points which cause the intersection are also known. Figure 10 below shows a sample demonstration of the process.

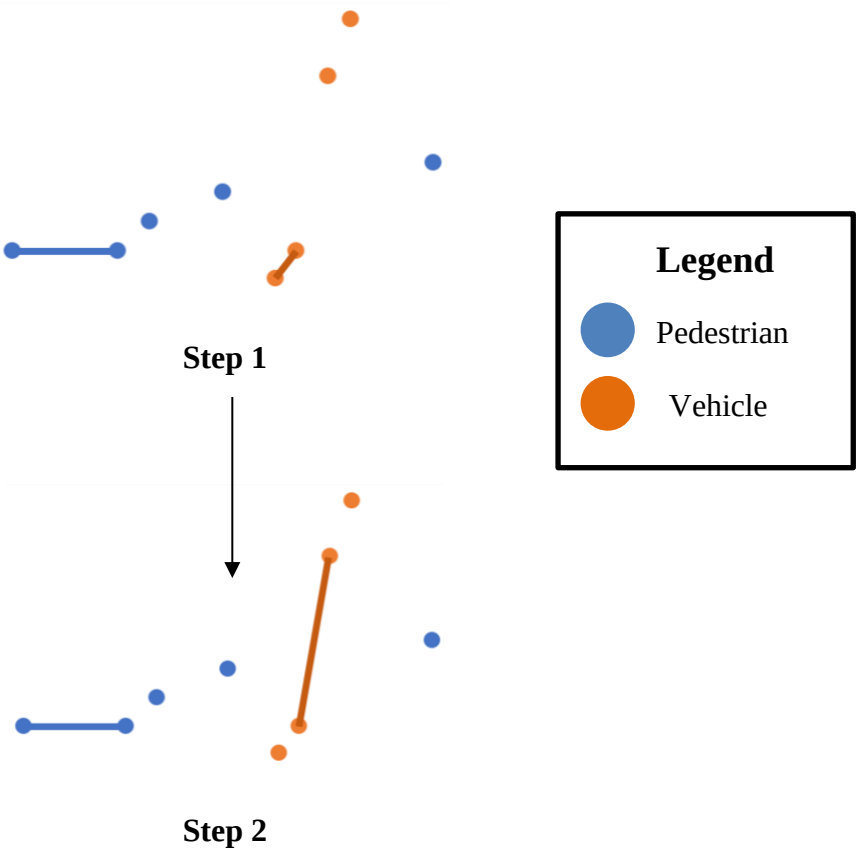


Figure 10 - Piecewise Intersection Step Through Steps 1 and 2

The software loops through each pedestrian segment and checks if an intersection occurs with each vehicle segment. If an intersection does not occur, the code checks if an intersection occurs with the next vehicle segment. If no intersection occurs on that

pedestrian segment, the process is repeated with the next pedestrian segment. The process continues until an intersection is found. Figure 11 below depicts the software finding an intersection in this sample.

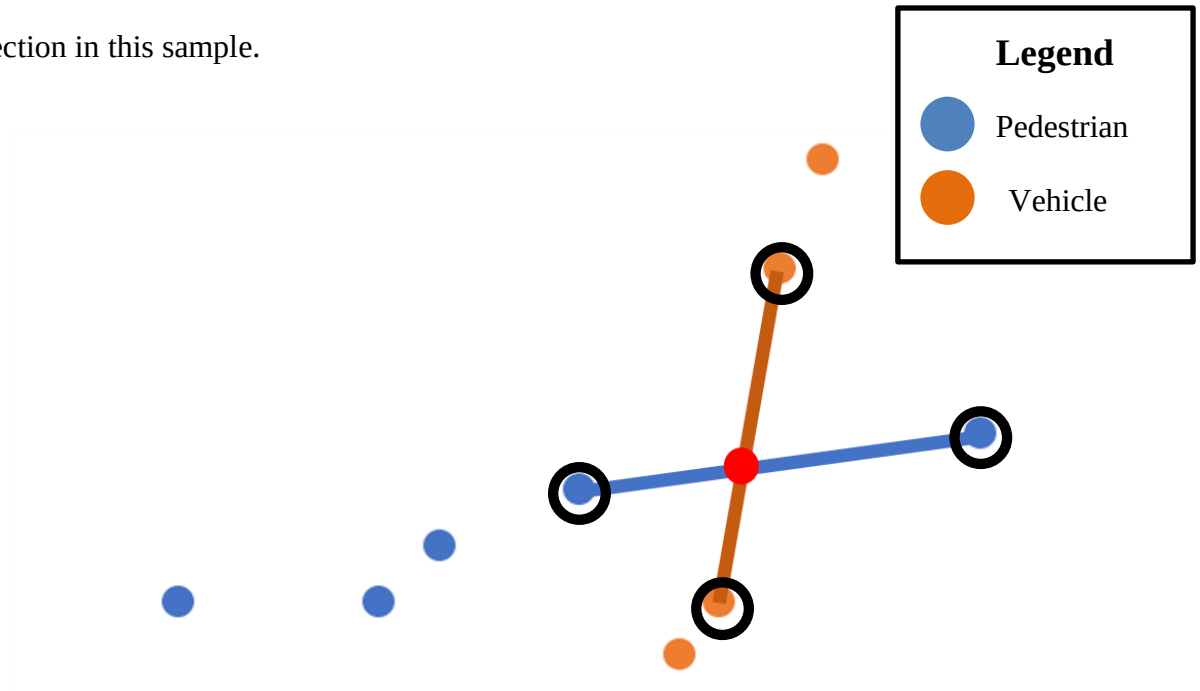


Figure 11 - Sample Found Intersection

In this case, the intersection occurred in step 13. The code checked for intersections with all previous pedestrian segments, but could not find one. The intersection occurs on the 4th pedestrian segment and 2nd vehicle segment. The Python package provides the (x,y) coordinates of the intersection point, and using this piecewise methodology, the points which cause the intersection are now known. With the four intersection points, highlighted with black circles, and their known timestamps, the time the vehicle and pedestrian are at the intersection point can be interpolated.

4.6.1 PET Calculation

Once the intersection point, and four points which cause the intersection are known. The average pedestrian and vehicle speed can be calculated. Since each point has a timestamp, the average speed is calculated by finding the distance between the two points and dividing by the time difference. The speed found does not represent any physical speed; it only represents the speed in pixel space. Once the pedestrian and vehicle segment speed is found, linear interpolation can occur to find the time each object was at the intersection point.

The linear interpolation assumes that the pedestrian and vehicle speed remains constant between each respective two points. While speed between two points may not be constant, the points are assumed to be within a close enough distance within which the speed should not vary drastically over the segment. To find the intersection point for each object, the distance between the first point which causes the intersection and the intersection point is calculated. Once the distance between the first point and intersection point is found, the time to intersection is found by dividing the distance by the speed, and adding the first point time to the intersection point. Figure 12 below shoes conceptually this calculation.

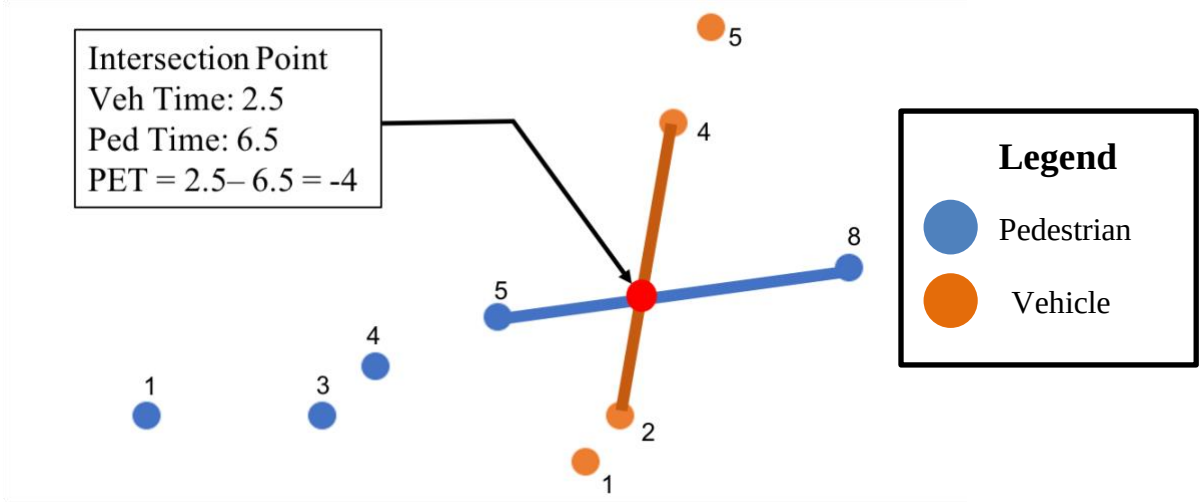


Figure 12 - Sample Intersection Time Calculation

From Figure 12, the pedestrian reaches the intersection point at approximately time 6.5, and the vehicle reaches the intersection at approximately time 2.5. The conflict software completes this process for each conflict. Once the intersection time for both the pedestrian and vehicle is found, the PET can be simply calculated using Equation 1 below.

Equation 1 - PET Calculation

$$PET = Vehicle\ Intersection\ Time - Pedestrian\ Intersection\ Time$$

With Equation 1, a negative PET means the pedestrian crossed behind the vehicle. A positive PET means the pedestrian crossed in front of the vehicle. A negative PET is a much less severe conflict compared to a positive PET. After the PET is calculated for each conflict, the results are saved to a .csv file. The PET and intersection point are used for visualization purposes which are described in CHAPTER 5.

4.7 Summary

This section demonstrates the method for filtering the data, identifying conflicts and intersections, and calculating the PET. The flowchart in Figure 13 summarizes the process.

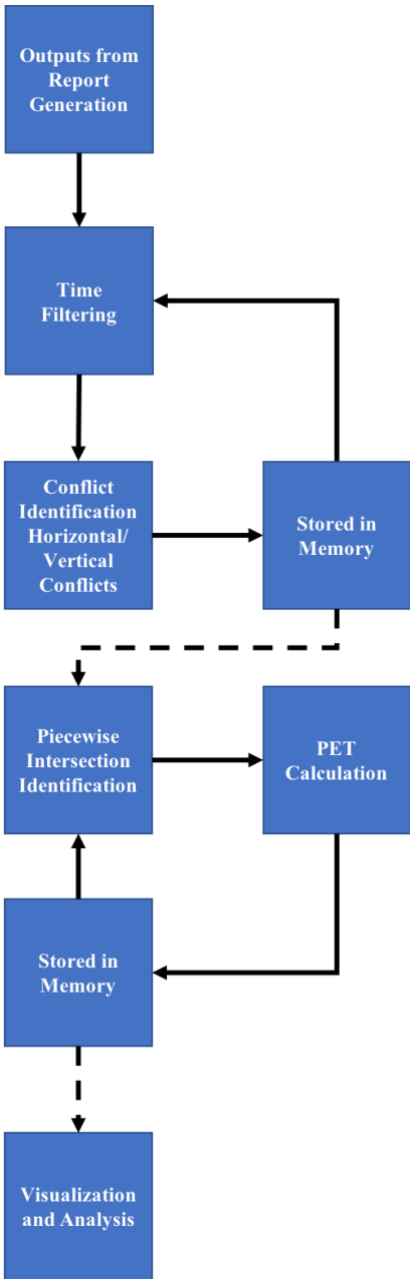


Figure 13 **- Conflict Identification and PET Calculation**

From Figure 13, the conflict detection software first imports the results from the report generation tool. The software then slices the data into the 10 second time window

described in Section 4.3. For each 10 second time window, the conflict software then begins conflict identification. Each conflict identified in a 10 second window is stored in memory. Once one 10 second window is complete, the conflict analysis steps one second and identifies conflicts in the next 10 second window. The dashed line represents that the intersection identification does not begin until the entire dataset is processed.

After all conflicts are identified, the intersection identification begins. By using the piecewise methodology described in Section 4.6, the conflict software identifies each intersection point. From identifying the intersection point, the pedestrian and vehicle intersection time is linearly interpolated. After linear interpolation, the PET is calculated using the simple formula in Equation 1. The process is repeated for each conflict. After calculating each conflict's PET, the visualization and analysis process begins.

CHAPTER 5. CONFLICT ANALYSIS RESULTS

5.1 Overview

This section discusses the results of the conflict analysis for four different sites. Additionally, this section describes the general outputs such as visualizations, and the file structure associated. This chapter will also discuss validation efforts and overall tracker accuracy.

5.2 Output Files and File Structure

As previously mentioned, the conflict analysis runs from the Reportgen outputs. To use the conflict detection software, the user must double click a desktop shortcut seen in Figure 14 which prompts the user to select the report folder.

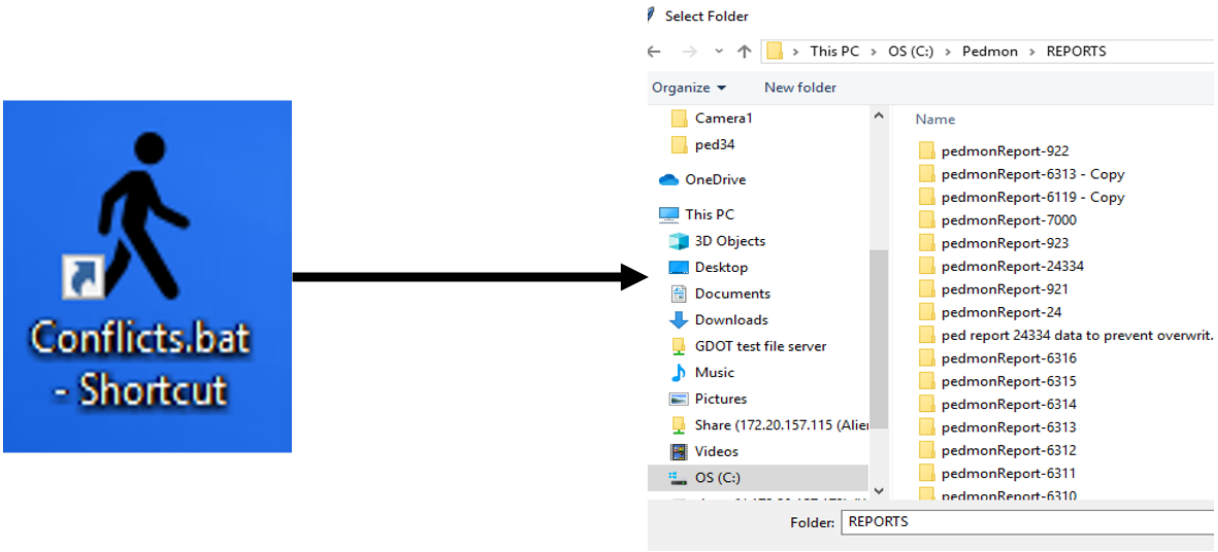


Figure 14 - Starting Conflict Analysis

Detailed instructions of operating the conflict analysis tool is in APPENDIX A. Conflict analysis software manual. This overview, however, provides the key file structure for the conflict detection software. The software saves new folders in the selected report folder which hold all the outputs. Figure 15 and Figure 16 depict this file structure.

Figure 15 - Output Folder Structure

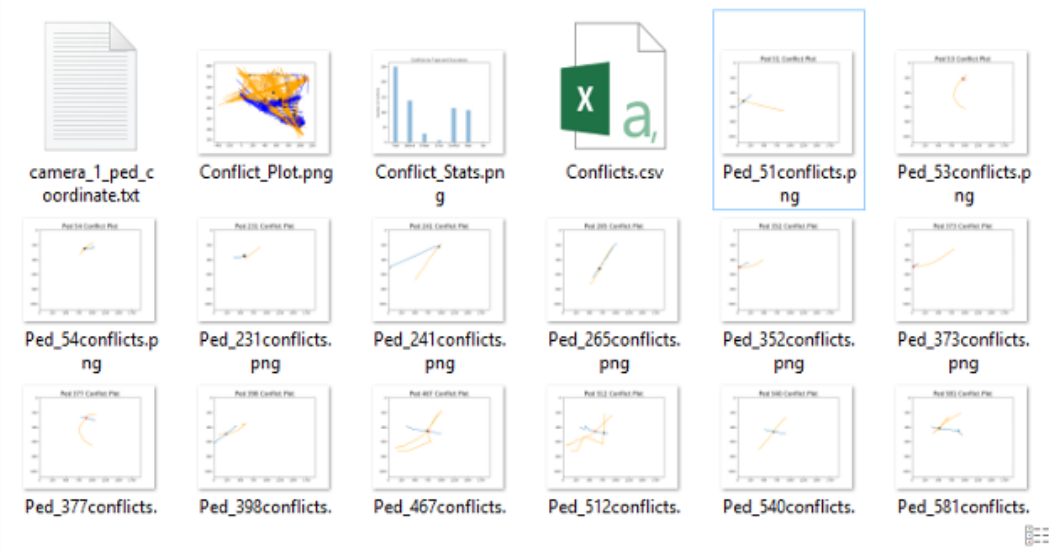


Figure 16 - Sample Outputs

Figure 15 shows the sample folder structure in a site report folder. In each conflict folder, subfolders exist with the outputs saved for each camera. General outputs which apply to all cameras include text files with overall statistics and percentages. The outputs

in each camera's subfolder are helpful with both validation and analysis. From Figure 16, each camera subfolder holds an overall conflict plot, individual conflict plots, general statistics, a conflict csv file, and a validation csv file.

Figure 17 shows a sample overall conflict plot. The orange lines represent vehicle tracks, and the blue lines represent pedestrian tracks. As visible in the picture, almost all conflicts occur horizontally, and a significant density exists in the same general area. Black dots on the conflict plot represent conflict points where pedestrians pass behind a vehicle, while red dots represent conflict points where pedestrians pass in front of a vehicle. A larger dot represents a more severe conflict – a PET closer to zero. The overall conflict figure provides quick high-level analysis of the conflicts that occur at a site.

Figure 17 - Sample Overall Conflict Plot with Size Scale

Along with plotting the entire conflict, each unique pedestrian's conflict plot is saved as well. Figure 18 below depicts a sample pedestrian conflict plot.

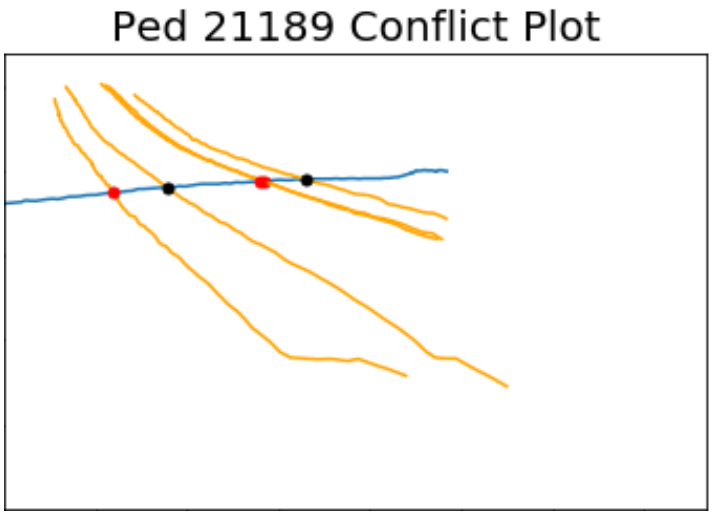


Figure 18 – Sample Individual Conflict Plot

Here, pedestrian 21189 conflicts with 5 vehicles while crossing the street. Out of these conflicts, three occur when the pedestrian is crossing in front of the vehicle, and two occur when a pedestrian is crossing behind a vehicle. These individual conflict plots provide insight into crossing locations, amount of conflicts per pedestrian, and assist in manual validations of conflicts.

Based on the types of conflicts and severity, general conflict statistics are generated with the conflict analysis as well. Figure 19 shows sample general conflict statistics. Overall, on this camera and site over 600 conflicts occurred. Most those conflicts were conflicts where the pedestrian passes behind the vehicle. For these statistics, the cutoff point for a conflict was +/- 10 seconds. A near conflict is between -3 and 3 seconds, while a severe conflict was between -1 and 1 seconds. For behind conflicts, a small percentage were near or severe, however, for conflicts where pedestrians cross in front of vehicles, a large percentage of those type of conflicts were found to be near conflicts.

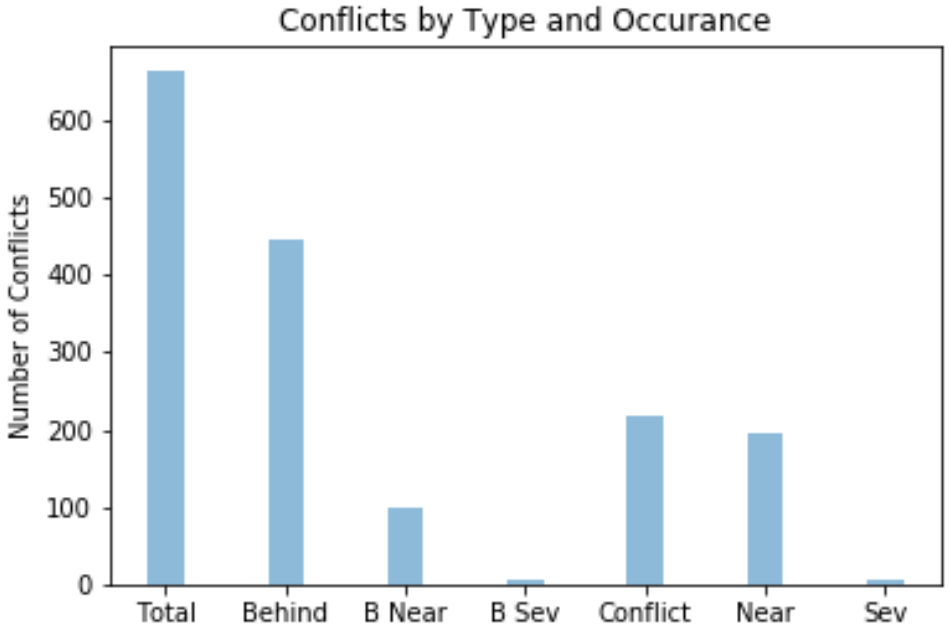


Figure 19 - Sample Conflict Statistics

The conflict outputs provide many useful datasets and statistics transportation engineers can use to analyze the type and severity of conflicts occurring on roadways. Additionally, the output file structure provides a quick and organized way to access outputs.

5.3 Detailed Site Analysis

This section discusses the four processed sites in detail. For each site, the camera views are depicted along with overlaid conflicts for selected camera.

5.3.1 Site 921: Buford Highway

In 2015, approximately three days of data were recorded on a segment of Buford Highway in Northeast Atlanta. The data were recorded using the four camera trailer system; the four camera views are seen in Figure 20 below.

Figure 20 - Site 921 Camera Views

As seen from the four camera views, crosswalks are not visible on this area of Buford Highway, so when pedestrians choose to cross, they must cross 7 lanes of 45 mph speed limit traffic unprotected. Table 1 below depicts the number of conflicts and conflict type for each camera.

Table 1. Site 921 Conflict Breakdown

Camera	Total Conflicts (-10 to 10 s)	Behind Conflicts (-10 to 0 s)	Behind Near Miss (-3 to 0 s)	Conflicts (0 to 10 s)	Near Miss (0 to 3 s)
1	368	184	41	184	152
2	336	234	50	102	96
3	140	83	21	57	52
4	664	447	98	217	195

From Table 1, the camera view with the highest number of conflicts is camera 4. The number of conflicts and conflict type can help understand an area's context; this view has a large amount of residences on one side disconnected from the MARTA stop, and other amenities. Without safe crossing locations nearby, pedestrians conflict with vehicles while mid-block crossing. Figure 21 below shows the overall conflict plot for camera 4.

Figure 21 - Site 921 Camera 4 Conflicts

Rather than walking to a crosswalk, people choose to cross directly. While most conflicts at this site are pedestrians which cross behind a car, nonetheless a significant amount of pedestrian—vehicle interactions occurred. Additionally, of conflicts where a pedestrian crossed in front of a car, a significant portion was categorized near miss – 3 seconds or less. One poorly timed gap, distracted pedestrian, or inattentive driver, would likely cause a fatality at 45 mph speeds.

5.3.2 Site 922: Joseph E. Lowery and Joseph E. Boone

Data was collected from this site over a period of 5 hours in November 2019. Figure 22 depicts the sample views from each camera. Similar to site 921, this site also used the trailer system. From the different views, cameras 1 and 4 are pointed at the intersection of Joseph E. Lowery and Joseph E. Boone while cameras 2 and 3 are directed away from the intersection.

Figure 22 - Site 922 Camera Views

While this site is a signalized intersection, conflicts between pedestrians and vehicles can still occur from permissive left and right turns, and from pedestrians disobeying the signal head. Table 2 shows that Camera 4 had the largest amount of conflicts with the number of conflicts being distributed almost equally between pedestrians crossing behind and in front of vehicles.

Table 2. Site 922 Conflict Breakdown

Camera	Total Conflicts (-10 to 10 s)	Behind Conflicts (-10 to 0 s)	Behind Near Miss (-3 to 0 s)	Conflicts (0 to 10 s)	Near Miss (0 to 3 s)
1	256	141	31	115	108
2	277	179	35	98	93
3	14	10	4	4	3
4	321	171	31	150	136

Figure 23 depicts all conflicts plotted for camera 4. From Figure 23, most conflicts occur when a pedestrian is within a crosswalk. While vehicles are unlikely to be traveling at high speeds, turning vehicles may not properly yield to pedestrians which generates conflicts.

Figure 23 - Site 922 Camera 4 Conflicts

In addition to camera 4, camera 2 also has many conflicts with 277. Figure 24 shows the conflict plot for this camera 2. Camera 2, like Buford Highway, has no crosswalks, yet a high number of conflicts. Watching the video, the construction area causes these conflicts; construction workers cross the street either to go to the bus stop or elsewhere. This information is useful to policy makers who can encourage developers to implement pedestrian—vehicle conflict mitigation efforts.

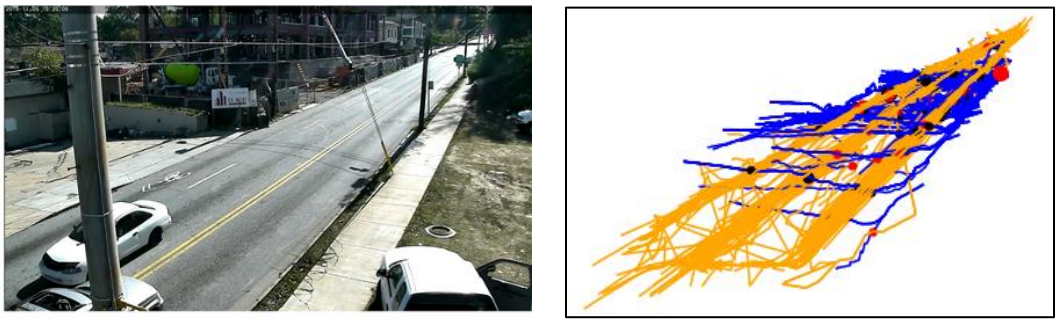


Figure 24 - Site 922 Camera 2 Conflicts

From Site 922, the conflict tool can highlight unexpected areas of conflict, such as a construction area, and provide information about pedestrian safety at intersections.

5.3.3 Site 923: Joseph E Lowery and MLK Boulevard

Along with site 922, site 923 was processed in November 2019 with about 5 hours of video saved. Figure 25 below shows the four camera views for site 922. Cameras 3 and 4 are directed at the intersection, and cameras 1 and 2 are pointed away from the intersection. Camera 2 specifically shows difficulty of situating the trailer in an urban environment; the gas station sign blocks most of this camera view.

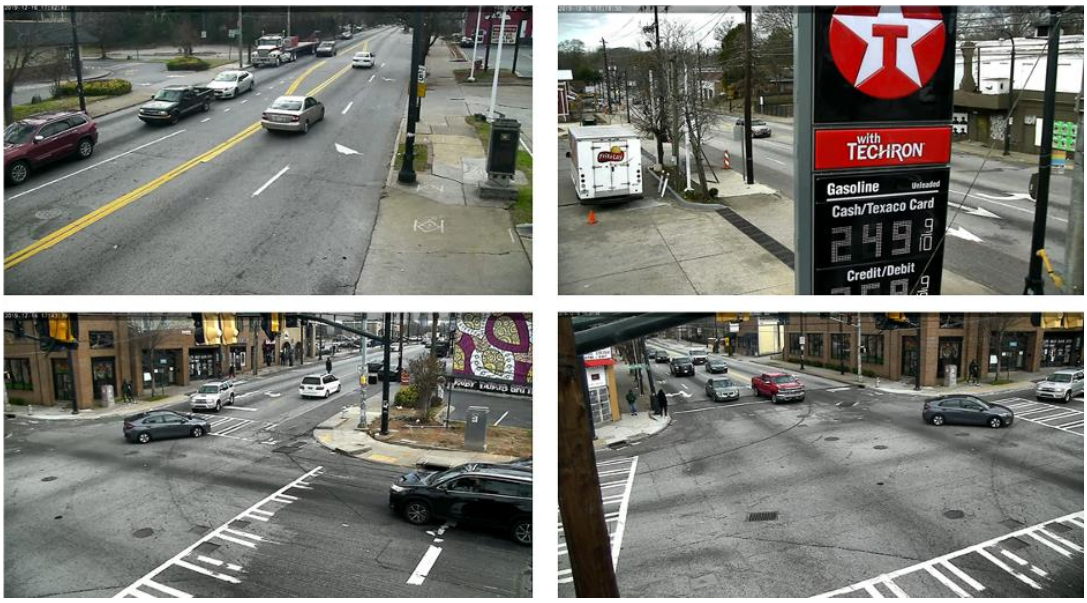


Figure 25 - Site 923 Camera View

Figure 26 shows the overall conflict plot of camera 4 – the view with the largest number of conflicts. Like Figure 23 of site 922, conflicts at this location are primarily in the crosswalk. This suggests that site 923 and site 922 are similar.

Figure 26 - Site 923 Camera 4 Conflicts

To minimize conflicts at an intersection, traffic engineers can change signal timing and provide pedestrian signal preemption.

5.3.4 Site 7000: 10th and Myrtle Street

This data was collected using a GoPro over a period of about a day. This data was collected before the installation of an RRFB signal at the same location, proceeded by the installation of a HAWK signal. Figure 27 shows a sample view from the GoPro and processed conflict overlay.

Figure 27 - Site 7000 Sample View and Conflicts

The conflict plot in Figure 27 show that most conflicts occur at the 10th Street crossing, as well as the driveway. Recording additional video at this intersection would provide data on how much has the pedestrian hybrid flashing signal has reduced conflict. By using this conflict analysis in before and after studies, the effectiveness of conflict analysis measures can be analyzed.

Table 3. Site 7000 Conflict Breakdown

Camera	Total Conflicts (-10 to 10 s)	Behind Conflicts (-10 to 0 s)	Behind Near Miss (-3 to 0 s)	Conflicts (0 to 10 s)	Near Miss (0 to 3 s)
1	606	270	31	336	289

Table 3 above shows the conflict analysis breakdown for site 7000. From the conflict analysis, over 600 conflicts occurred in a day. Most of these conflicts were conflicts that pedestrians walked in front of a vehicle, and , a significant number of these conflicts are classified as near where a vehicle crosses a pedestrian's path in 3 seconds or fewer.

5.3.5 Site Analysis Summary

Overall, conflict analysis has been performed on 4 different sites, and almost 12 days of video. Each site has its own context that a transportation engineer must incorporate in the conflict analysis. For example, Buford Highway and 10th and Myrtle are unsignalized sites with right angle conflicts, while Joseph E. Lowery and Joseph E. Boone are focused on intersections with many conflicts which are primarily caused by turning vehicles. The conflict analysis allows GDOT to empirically compare different locations and proactively implement conflict mitigation measures.

One method of comparing different sites includes analyzing the exposure rate. Since different sites have different number of pedestrians crossing and context, analyzing the exposure rate provides a normalized comparison of different sites and camera views. Table 4 shows the total number of pedestrians, conflicts, and conflict percentage (total conflicts/total peds) for each site and camera view.

Table 4. Site Conflict Percentages and Context

Site ID	Camera ID	Total Pedestrians	Total Conflicts	Conflict %	View Context
921	1	955	368	39%	Midblock zoom
921	2	789	336	43%	Midblock far
921	3	749	140	19%	Camera 1 far
921	4	653	664	102%	Camera 2 zoom
922	1	715	256	36%	Intersection
922	2	314	277	88%	Construction Midblock
922	3	113	14	12%	Midblock near intersection
922	4	1243	321	26%	Intersection
923	1	365	367	101%	Midblock
923	2	1055	55	5%	Midblock near intersection
923	3	1619	721	45%	Intersection
923	4	2215	943	43%	Intersection
7000	1	2879	613	21%	Unsignalized Intersection

Based on the table above, different camera views and sites have varying conflict percentages. For site 921, both zoomed cameras have a higher conflict percentage and total number of conflicts than the far views. These results suggest that more meaningful results will come from a targeted view with the highest amount of pedestrians crossing. As the distance increases in the view, pedestrians become more difficult to track.

Surprisingly, the number of conflicts on some sites is greater than the actual number of pedestrians. Looking at view context and conflict percentage shows that a relationship between the two exist. The highest three conflict percentages occurred at midblock

crossings that were not close to an intersection. The lowest two conflict percentages occurred at a midblock crossing, but those crossings were close to an intersection.

From this analysis, proximity to an intersection discourages midblock crossings. Additionally, midblock crossings not near intersections have high conflict rates. This relationship also is intuitive; a person crossing midblock may conflict with multiple cars in each direction. Alternatively, a person crossing at an intersection would likely only conflict with one or two cars that are turning permissively.

This comparison provides GDOT with insights on which areas need infrastructure changes. Rather than reacting to pedestrian—vehicle crashes or fatalities, the automated video processing and conflict analysis provides a proactive approach to prevent accidents.

5.4 Validation

This section describes the process of validating the conflict detection software. To provide a meaningful, usable tool, the conflict software results must be accurate. If the conflict software is not detecting actual conflicts, and primarily detecting false positives, the tool will not provide meaningful results. Additionally, the PET that the tracker outputs must be accurate. If not, the severity of a conflict cannot be measured. To assess the conflict code's accuracy, manual validation occurred on the Site 921 dataset.

5.4.1 False Positive Analysis

To conduct false positive analysis, the conflict analysis was integrated with the video playback tool. With the video playback tool, each conflict was played back in the video, and this allowed for quick manual validation. Rather than match almost 10 days of video,

each conflict could be jumped to in the video. Jumping though each conflict, the conflicts are categorized as true conflicts or false positives.

Table 5 shows the outcome of the manual validation. The manual validation occurred on over 1500 conflicts with 89% of these conflicts being actual conflicts. Of the false positives, 26% were caused by motorcycles. Ultimately, the tracker software tracks people and currently does not differentiate between those on motorcycles and those on foot. Because the software labels the motorcycle rider as a pedestrian and the motorcycle itself as a vehicle, the vehicle and pedestrian trajectories inevitably cross and therefore conflict. Previously, almost 50% of false positives were causes by motorcycles. Logic was added to the tracker to help filter out pedestrians on motorcycles, and this filtering cut the motorcycle false positive rate in half.

Table 5. Real Conflicts and False Positives on Site 921

	Camera 1	**Camera 2**	**Camera 3**	**Camera 4**	**Total**	**%**
Real Conflicts	270	320	128	622	1340	**89%**
False Positives	98	16	12	42	168	**11%**

Other causes of false positives include turning vehicles, pedestrians on the sidewalk, and headlights. Table 6 shows the breakdown of false positives by type. An accuracy of 89% demonstrates the conflict software's validity in identifying real conflicts. While some false positives exist, these can be filtered through quickly through manual validation.

Table 6. False Positive Analysis

	Camera 1	Camera 2	Camera 3	Camera 4	Total	%
Motorcycle	23	7	7	7	44	**26%**
Turning	31	0	0	0	31	**18%**
Object on car	6	1	0	0	7	**4%**
On Sidewalk	0	1	1	18	20	**12%**
Headlight	16	0	3	4	23	**14%**
Headlight-B&W	7	6	1	0	14	**8%**
Other	15	1	0	13	29	**17%**

Through validation, additional improvements to the conflict software can be identified. For example, future additional logic can be built in the tracker to identify and filter false conflicts. The initial conflict detection software nonetheless depicts a high accuracy, and false positives which do exist can be quickly categorized by manual validation.

5.4.2 PET Validation

In addition to overall conflict detection, accurate PET are also critical to the software's validity. The PET validation followed a similar process to the accuracy validation, and also used the video playback tool. The following workflow was established for PET validation:

1. *For a certain camera, load up the video playback tool.*

2. *Open the individual conflict plots for each unique ped.*
3. *As you jump through the playback tool, match the pedID with the individual conflict plot.*
4. *Identify the actual PET time for each conflict*.
5. *Input into the validation.csv spreadsheet.*

This process was completed for 764 true conflicts on Site 921. Table 7 shows the average differences between the true PET, labeled ground truth (GTr) and software PET. Overall, the conflict software PET outputs are within a tenth of a second of the real PET. The conflicts are broken down into both Away and Toward categories as well as Front and Behind categories. The Away and Toward categorizations represent whether the vehicle was traveling away or toward the camera. The Front and Behind categories represent whether the pedestrian walked behind or in front of the vehicle.

Table 7. Average Difference in Detection PET vs. Ground Truth

	Cam1	**Cam 2**	**Cam3**	**Cam 4**	**Average**
All Conflicts	0.14	0.01	0.10	0.11	**0.09**
All Behind Conflicts	-0.05	-0.07	0.09	0.09	**0.00**
All Front Conflicts	0.35	0.17	0.11	0.14	**0.21**
All Away Conflicts	0.36	0.40	0.20	0.27	**0.33**
All Toward Conflicts	-0.05	-0.08	-0.03	0.00	**-0.04**
Away & Behind	0.08	0.18	0.16	0.24	**0.17**

Away & Front	0.81	0.75	0.30	0.33	**0.59**
Toward & Behind	-0.19	-0.11	-0.07	0.00	**-0.09**
Toward & Front	0.08	0.00	-0.01	0.00	**0.02**

Breaking down the groups, the largest difference occurs in the Away categories. Away & Behind conflicts have an average difference of 0.17 seconds, while Away & Front conflicts have an average difference of 0.59 seconds. The location where the trajectories are drawn causes these differences. By default, both vehicle and pedestrian trajectories are drawn at the middle-bottom of the shapes bounding box.

For a car traveling away from the camera, that means the back of the car intersects the pedestrians path, not the front. For cases where a pedestrian walks in front of a vehicle (positive PET), the conflict tool detected outputs on average 0.59 seconds greater that what actually occurred. This means a 4.59 PET detected Away & Front conflict was actually a PET of 4.0 seconds. This difference is not necessarily insignificant, and the trajectory correction section expands on ways to account for vehicles traveling away.

5.4.3 Trajectory Correction

The bottommost point of the bounding box is accurate for all pedestrians, since it represents their feet in all cases. For cars, this assumption is not accurate. Cars going toward the frame will have their bounding boxes represented accurately, since it will be the point just below the hood of the car. Turning vehicles, and cars going away from the frame are therefore not accurately represented. Turning vehicles should have their trajectories on the right or left of the bounding box, depending on the type of turn and direction they are

travelling. Cars travelling in a straight line away from the frame should have their trajectories closer to the top of the bounding box.

To accurately reflect conflicts between pedestrians and vehicles, additional logic was included in the conflict software. The logic looks at the direction of each unique vehicle, and for those vehicles traveling away, the trajectory is adjusted over its entire course. After initial tests, the trajectory adjustments were completed the 2.5 day dataset. Since the code searches for all vehicles tracked over 4 cameras and rewrites those tracks in the database, the calculation is computationally intensive. For this 2.5 day dataset, this computation took 4 days. The speed of this portion of code can likely be improved, however, dropping conflicts for only about a 0.5 second improvement on away conflicts may not be worth it. Nonetheless, the trajectory correction code is commented out and exists for future testing.

5.4.4 Validation Summary

Overall, this initial validation set demonstrates that the conflict detection performs reasonably well in both identifying actual conflicts and PET. While some false positives occur, 89% of conflicts were identified as real conflicts. Additionally, PET on average is within 0.1 second accurate. The main differences occur with vehicles travelling away due to the placement of their trajectories. Trajectory correction logic exists in the software, however, it is currently too computationally intensive.

5.5 Tool Integration

To provide GDOT with a meaningful tool, the purpose of this project was to provide repeatable conflict analysis. Rather than an individual case study and proof of concept, the conflict analysis tool was required to work within the existing software suite.

5.5.1 Video Playback Integration

In addition to all analysis-oriented outputs, the conflict code also generates a csv file which is compatible with the video playback tool. Using the video playback tool with the conflict analysis outputs, a user can jump to the time and location where a conflict occurs for a pedestrian. This video playback compatibility not only allows for ease of manual validation, but also provides compelling video of people crossing the street. Rather than just a black box with conflict statistics, the integration with the video playback tool provides the ability to jump to conflicts and watch how road users interact. Watching this interaction allows traffic engineers to directly visualize the site's safety, which the data alone cannot provide.

5.6 Results Summary

The conflict analysis software has currently been processed on almost 11 days of video. The conflict analysis accurately identifies conflicts and its respective PET at both intersection and non-intersection locations. Additionally, the integration with the video playback tool provides traffic engineers the ability to play back conflicts in video. The structure of this software, and the existing tools from GTRI, provides a flexible system that can be applied with different video feeds and at different sites.

CHAPTER 6. CONCLUSION

This thesis demonstrates a practical software that GDOT can implement. Using automatic detection paired with the conflict analysis, GDOT can identify and compare unsafe corridors. These tools allow GDOT to be proactive in selecting location for mitigation efforts rather than being reactive and using crash data. The portable trailer system allows GDOT transportation engineers to select existing potential troublesome areas if video does not already exist. If video exists from a different source, GDOT can use that video to detect conflicts.

The simple structure of these tools allows for transportation engineers to simply load up a video and process; no additional calibration is needed. Rather than watch hours of video to count pedestrians and conflicts, the software instead analyzes the video which allows transportation engineers more time to analyze results. From the results of the Multimon Reportgen, the detection accuracy is quite high. Additionally, the ability to playback and quickly manually validate pedestrian counts provides traffic engineers with an important tool.

The conflict detection software has a high accuracy level for both detections and PET. The overall accuracy for the conflict detection is about 90%. For true conflicts, the PET calculations averaged within 0.1 seconds of real conflicts. While cars travelling away from the scene had more a higher post-encroachment difference, this difference was still only 0.5 seconds. This accuracy and precision provides confidence in the conflict analysis outputs. While some false positives still exist, the ability to playback conflicts in the video provides quick manual validation. A three-day dataset that would previously take 72-hours

for an engineer to analyze conflicts, can now take less than an hour by jumping through the video. This thesis provides traffic engineers a practical and easy-to-use tool to improve pedestrian safety.

APPENDIX A. CONFLICT ANALYSIS SOFTWARE MANUAL

This appendix discusses the steps to use the conflict analysis software.

A.1 Description

The Conflict Analysis Tool is the program for identifying conflicts from the pedestrian tracking program. This program is designed to be run after the Report Generation Tool. This tool allows a user to specify the report folder, and outputs pedestrian-vehicle conflicts and statistics. This manual describes those processes in detail. Figure 28 shows the conflict shortcut which is on the desktop.

Figure 28 - Conflict tool shortcut

A.2 Conflict.bat Shortcut

Double clicking the Conflicts.bat-Shortcut in Figure 28, located on the desktop, begins the conflict analysis software. Double clicking shortcut opens a console terminal window and prompts the selection of a Report directory.

A.3 Select Report Directory

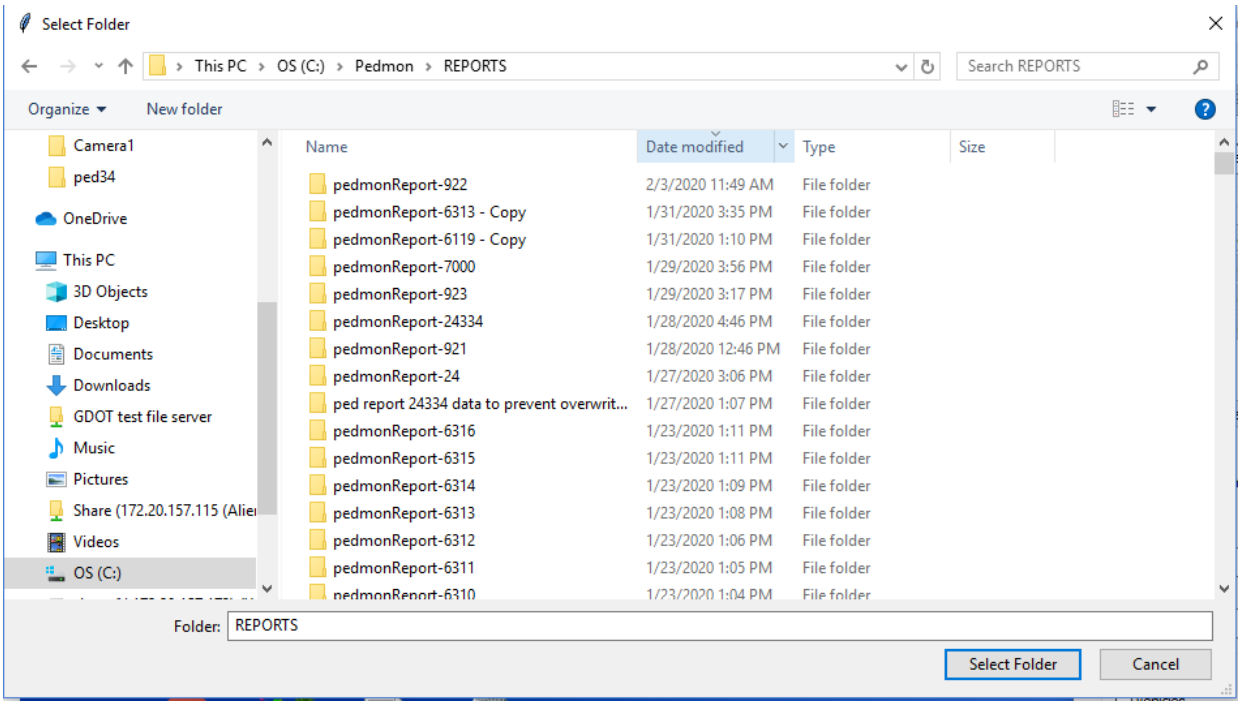


Figure 29 - Select report directory

Figure 29 shows the select the report directory which the conflict analysis will be run on. Once the Report folder is selected, the user will be prompted whether they want to analyze behind conflicts, and the conflict window to analyze. A sample report display is seen in Figure 29.

A.4 Terminal Feedback

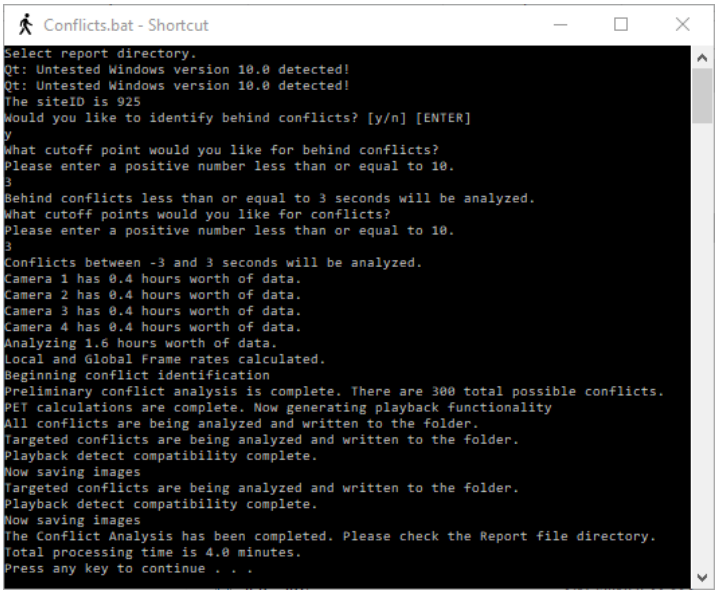


Figure 30 - Terminal feedback

While the conflict analysis is processing, the code provides feedback allows the user to understand what portion of the analysis is currently running. Figure 30 shows sample terminal feedback.

A.5 Steps for Running Conflict Analysis

1. Double click the desktop file shortcut seen in Figure 1.

2. Select the report directory to process on.

3. Choose whether you want behind conflicts analyzed or not by typing “y” or “n” and hitting enter.

4. Choose the cutoff times for behind and front conflicts by entering a positive number less than or equal to 10.

5. Let the conflict analysis run. Depending on the length of the report, runtime may be significant. Check terminal feedback to see updates on the conflict process.

6. The feedback in Figure 3 shows a sample analysis for conflicts between -3 and 3 seconds.

A.6 Conflict Analysis Outputs

The conflict analysis tool creates a Conflicts-all folder in the selected report directory, and the targeted conflicts as well. For example in the figure below, the Conflicts-all folder exists, and the Conflicts--5-5 is the analysis with all conflicts between -5 and 5 seconds. Creating and saving all conflicts allows for faster processing of different time thresholds. Figure 31 shows the sample folder outputs.

Figure 31 - Sample folder outputs

Under the Conflict subfolder of the report, folders are created for each camera view. Each camera subfolder has detailed data for each camera view which is seen in Figure 32. Additionally, general conflict data is located in the .txt files in the main subdirectory, seen below.

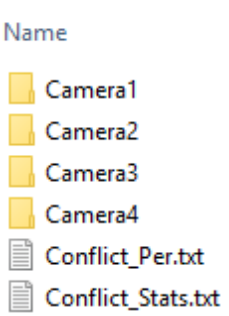


Figure 32 – Sample general conflict outputs

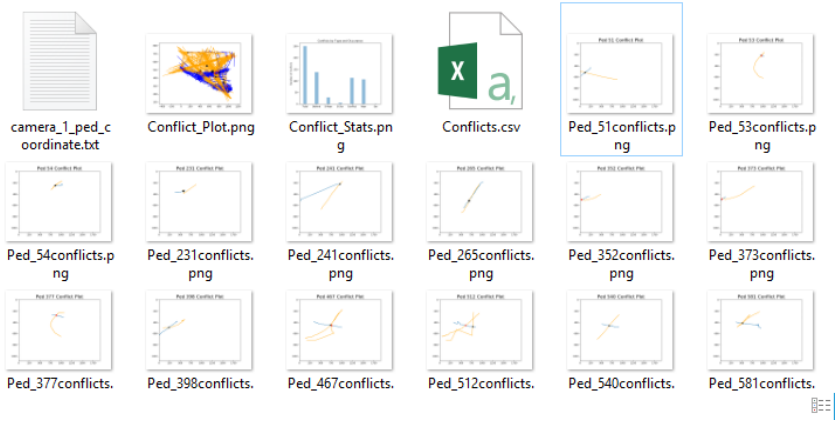


Figure 33 - Sample camera outputs

For each camera, the conflict analysis outputs an overall conflict plot, conflict stats graph, lists all conflicts in a csv file, and has each pedestrians' conflicts saved as an image.

Figure 33 depicts a sample layout. Additionally, the coordinate.txt file is used for compatibility with the Playback Detection Tool.

A.7 Playback Detection Compatibility

The Conflict Analysis Tool has automatic compatibility with the Playback Detection Tool. To jump to conflicts in the video, start the Playback Detection Tool and follow the user manual. The targeted conflicts folder (Conflicts--5-5) will only depicts conflicts between the cutoff points.

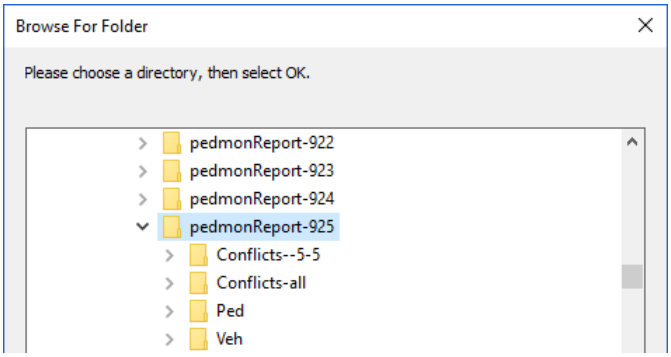


Figure 34 - Playback detection compatibility

When prompted to select a report directory in the Playback Detection Tool, select the Conflicts directory, seen in Figure 34, on the desired site. The Playback Detection Tool will now display the conflicts in video with the ability to jump to different conflicts.

APPENDIX B. CONFLICT ANALYSIS SOFTWARE CODE

The following appendix depicts all the code used for this thesis. Code that is commented out begins with a "#". This software is coded in Python, and also shows the required software packages.

B.1 Code

```
# coding: utf-8

#Importing all the necessary packages

import pandas as pd
import numpy as np
import shapely
import sys
from shapely.geometry import LineString, Point
from shapely import wkt
import timeit
import math
import os
import matplotlib.pyplot as plt
import datetime
import statistics
import tkintr as tk
from tkinter import filedialog
import warnings

warnings.filterwarnings("ignore", category=RuntimeWarning)
pd.options.mode.chained_assignment = None

#print(shapely.__version__)
root = tk.Tk()
root.withdraw()

print('Select report directory.')

file_path = filedialog.askdirectory()
dummy_str = 'C:/Pedmon/REPORTS/pedmonReport-'

if dummy_str in file_path:
    site = file_path.replace(dummy_str,'')
    print('The siteID is ' + site)
else:
```

```
    print('Improper report folder. Folder should be written
as pedmonReport-')
    sys.exit()


siteID = int(site)


loop_flag = True

while loop_flag:

    behind = input('Would you like to identify behind
conflicts? [y/n] [ENTER]\n')
    if ('y' in behind) or ('Y' in behind):
        behind_flag = True
        loop_flag = False

        cflag = True
        while cflag:
            bc = input('What cutoff point would you like
for behind conflicts?\nPlease enter a positive number less
than or '
                       'equal to 10.\n')
            if bc.isdigit():
                if (int(bc) <= 10) & (int(bc)>0):
                    cflag = False
                    behind_cutoff = int(bc)
                else:
                    print('Please enter a positive number
less than or equal to 10.')
            else:
                print('Please enter a number.')

        print('Behind conflicts less than or equal to ' +
bc + ' seconds will be analyzed.')
    elif  ('n' in behind) or ('N' in behind):
        behind_flag = False
        loop_flag = False
        bc = str(0)
        print('Behind conflicts will be ignored.')
    else:
        print('Please respond with "y" or "n".\n')

loop_flag = True
```

```
while loop_flag:

    ccut = input('What cutoff points would you like for
conflicts?\nPlease enter a positive'
                         ' number less than or equal to
10.\n')

    if ccut.isdigit():
        if (int(ccut) <= 10) & (int(ccut)>0):
            loop_flag = False
            conflict_cut = int(ccut)
        else:
            print('Please enter a positive number less than
or equal to 10.')
    else:
        print('Please enter a number.')

if int(bc) > 0:
    print('Conflicts between -' + bc + ' and ' + ccut + '
seconds will be analyzed.')

else:
    print('Conflicts less than or equal to ' + ccut + '
seconds will be analyzed.')



pedfolder = file_path +"\\" + "Ped"
vehfolder = file_path + "\\" + "Veh"

dirpath = file_path + "\\" + "Conflicts-all"

if bc == str(0):
    secondpath = file_path +"\\" + "Conflicts-" + bc + "-"
+ ccut
else:
    secondpath = file_path + "\\" + "Conflicts--" + bc + "-
" + ccut


####If the all conflicts have already been run, then
conflict identification does not need to occur again
####This means that the conflict csv file already possesses
all the existing information
####Need to build in code can read in the conflicts csv
file of the all conflicts and identify places from there
```

```
all_flag = False

try:
    os.mkdir(dirpath)
except:
    all_flag = True
cam1 = dirpath + "\\" + "Camera1"
cam2 = dirpath + "\\" + "Camera2"
cam3 = dirpath + "\\" + "Camera3"
cam4 = dirpath + "\\" + "Camera4"

try:
    os.mkdir(cam1)
except:
    cam1all = pd.read_csv(cam1 + '\\Conflicts.csv')
try:
    os.mkdir(cam2)
except:
    cam2all = pd.read_csv(cam2 + '\\Conflicts.csv')
try:
    os.mkdir(cam3)
except:
    cam3all = pd.read_csv(cam3 + '\\Conflicts.csv')
try:
    os.mkdir(cam4)
except:
    cam4all = pd.read_csv(cam4 + '\\Conflicts.csv')

if all_flag:
    allconflicts = pd.concat([cam1all, cam2all, cam3all,
cam4all], join='outer', ignore_index=True)

try:
    os.mkdir(secondpath)
except:
    print("Folder already exists")
cam1 = secondpath + "\\" + "Camera1"
cam2 = secondpath + "\\" + "Camera2"
cam3 = secondpath + "\\" + "Camera3"
cam4 = secondpath + "\\" + "Camera4"

try:
    os.mkdir(cam1)
except:
    print("Folder already exists")
```

```
try:
    os.mkdir(cam2)
except:
    print("Folder already exists")
try:
    os.mkdir(cam3)
except:
    print("Folder already exists")
try:
    os.mkdir(cam4)
except:
    print("Folder already exists")



fps = 30
time = 10 * fps
# In[2]:


# #Different reports will be generated on different siteIDs
# siteID = 923
# #This will be adjusted since the current folder structure
is not ideal
# folder = r"C:\Pedmon\REPORTS\pedmonReport-" + str(siteID)
# pedfolder = folder + "\\" + "Ped"
# vehfolder = folder +"\\" + "Veh"

#This file structure assumes that all four cameras had
reports run on them, will need to add in try/except

#Reading in all the ped df since the current code is camera
based


start_time = timeit.default_timer()

if not all_flag:

    try:
        ped1 = pd.read_csv(pedfolder +
"\Camera1\camera_1_detailed_report.txt")
    except:
        ped1 = pd.DataFrame()

    if len(ped1) > 0:
```

```
        ped1['Cam_ID'] = 1
        # print(len(ped1))
    try:
        ped2 = pd.read_csv(pedfolder +
"\Camera2\camera_2_detailed_report.txt")
    except:
        ped2 = pd.DataFrame()

    if len(ped2) > 0:
        ped2['Cam_ID'] = 2
    # print(len(ped2))

    try:
        ped3 = pd.read_csv(pedfolder +
"\Camera3\camera_3_detailed_report.txt")
    except:
        ped3 = pd.DataFrame()

    if len(ped3) > 0:
        ped3['Cam_ID'] = 3
    # print(len(ped3))

    try:
        ped4 = pd.read_csv(pedfolder +
"\Camera4\camera_4_detailed_report.txt")
    except:
        ped4 = pd.DataFrame()

    if len(ped4) > 0:
        ped4['Cam_ID'] = 4
    # print(len(ped4))

    # Same for veh df
    try:
        veh1 = pd.read_csv(vehfolder +
"\Camera1\camera_1_detailed_report.txt")
    except:
        veh1 = pd.DataFrame()

    if len(veh1) > 0:
        veh1['Cam_ID'] = 1
    # print(len(veh1))

    try:
        veh2 = pd.read_csv(vehfolder +
"\Camera2\camera_2_detailed_report.txt")
```

```
    except:
        veh2 = pd.DataFrame()

    if len(veh2) > 0:
        veh2['Cam_ID'] = 2
        # print(len(veh2))

    try:
        veh3 = pd.read_csv(vehfolder +
"\Camera3\camera_3_detailed_report.txt")
    except:
        veh3 = pd.DataFrame()

    if len(veh3) > 0:
        veh3['Cam_ID'] = 3
    # print(len(veh3))

    try:
        veh4 = pd.read_csv(vehfolder +
"\Camera4\camera_4_detailed_report.txt")
    except:
        veh4 = pd.DataFrame()

    if len(veh4) > 0:
        veh4['Cam_ID'] = 4
        # print(len(veh4))

    # In[3]:


    # Combining the pedestrian and veh df
    ped_com = pd.concat([ped1, ped2, ped3, ped4],
join='outer', ignore_index=True)
    veh_com = pd.concat([veh1, veh2, veh3, veh4],
join='outer', ignore_index=True)

    if len(ped_com) == 0:
        print('No pedestrian report exists in this
directory. Please run a pedestrian report on this site
ID.')
        sys.exit()

    elif len(veh_com) == 0:
        print('No vehicle report exists in this directory.
Please run a vehicle report on this site ID.')
        sys.exit()
```

```
    # Adds a global frame to both dataframes
    ped_com['globalframe'] = ped_com['frame_id']
    veh_com['globalframe'] = veh_com['frame_id']

    # Adds a videonum column to both dataframes
    veh_com['videonum'] = 0
    ped_com['videonum'] = 0


    ######Outputs some data about the amount of vehicles
processing#######

    cam = 1
    time_dif = 0

    while cam <= 4:
        x = 0
        if len(veh_com.loc[veh_com['Cam_ID'] == cam]) > 0:
            cam_veh = veh_com.loc[veh_com['Cam_ID'] == cam]
            max_time = cam_veh.frametime.max()
            min_time = cam_veh.frametime.min()
            maxtime = datetime.datetime.strptime(max_time,
'%m/%d/%Y %I:%M:%S %p')
            mintime = datetime.datetime.strptime(min_time,
'%m/%d/%Y %I:%M:%S %p')
            x = maxtime - mintime
            time_dif = round(x.total_seconds() / 3600, 1) +
time_dif

            if round(x.total_seconds() / 3600, 1) <= 24:
                print('Camera ' + str(cam) + ' has ' +
str(round(x.total_seconds() / 3600, 1)) + ' hours worth of
data.')
            else:
                print('Camera ' + str(cam) + ' has ' + str(
                    round(x.total_seconds() / 3600 / 24,
1)) + ' days worth of data.')
        cam = cam + 1

    if time_dif <= 24:
        print('Analyzing ' + str(time_dif) + ' hours worth
of data.')

    else:
```

```
        time_dif = round(time_dif / 24, 1)
        print('Analyzing ' + str(time_dif) + ' days worth
of data.')

    # In[84]:


    # Gets the local and global frame rate for
playback_detect integration

    rolling = pd.DataFrame(columns=['Video', 'Total',
'Cam_ID'])

    cam = 1

    while cam <= 4:

        # print(cam)

        # Gets one cameras worth of pedestrian data
        ped_onecam = ped_com.loc[ped_com['Cam_ID'] == cam]
        fileuni = ped_onecam.filename.unique()

        # Creates an empty dataframe
        global_df = pd.DataFrame(columns=['Video', 'Video
Num', 'Max', 'Cam_ID'])

        # Gets one cameras worth of vehcile data
        veh_onecam = veh_com.loc[veh_com['Cam_ID'] == cam]
        vfileuni = veh_onecam.filename.unique()

        # Checking to see which df has more videos in it
        if len(fileuni) > len(vfileuni):
            fileuni = fileuni
        else:
            fileuni = vfileuni

        # Loops through the one with the larger number of
files
        for i in range(len(fileuni)):
            video = fileuni[i]

            # i starts at 0
            num = i + 1
```

```
            # Filtering down specific dataframes
            pfilt = ped_com.loc[(ped_com['filename'] ==
video) & (ped_com['Cam_ID'] == cam)]
            vfilt = veh_com.loc[(veh_com['filename'] ==
video) & (veh_com['Cam_ID'] == cam)]

            # Checking to see whether the ped or vehicle
max frame is longer
            if pfilt.frame_id.max() > vfilt.frame_id.max():
                mframe = pfilt.frame_id.max()
            else:
                mframe = vfilt.frame_id.max()

            # Adding this data to the global df
            global_df = global_df.append({'Video': video,
'Video Num': num, 'Max': mframe, 'Cam_ID': cam},
                                         ignore_index=True)

        # Loops through the global_df to get the specific
video numbers added to the original ped/veh df
        for i in range(len(global_df)):
            video = global_df['Video'][i]
            num = global_df['Video Num'][i]

            slc = ped_com.loc[(ped_com['filename'] ==
video) & (ped_com['Cam_ID'] == cam)]
            slc['videonum'] = num
            ped_com.loc[(ped_com['filename'] == video) &
(ped_com['Cam_ID'] == cam)] = slc

            vslc = veh_com.loc[(veh_com['filename'] ==
video) & (veh_com['Cam_ID'] == cam)]
            vslc['videonum'] = num
            veh_com.loc[(veh_com['filename'] == video) &
(veh_com['Cam_ID'] == cam)] = vslc

        # Loops through global_df again
        for i in range(len(global_df)):
            num = global_df['Video Num'][i]
            mframe = global_df['Max'][i]

            # Adding a global frame rate to the ped_com
dataframe
            slc = ped_com.loc[(ped_com['videonum'] > num) &
(ped_com['Cam_ID'] == cam)]
```

```
            slc['globalframe'] = slc['globalframe'] +
mframe
            ped_com.loc[(ped_com['videonum'] > num) &
(ped_com['Cam_ID'] == cam)] = slc

            # Adding a global frame rate to the vehicle_com
dataframe
            vslc = veh_com.loc[(veh_com['videonum'] > num)
& (veh_com['Cam_ID'] == cam)]

            # Adding the max frame rate to all videos after
the one where the max frame occurs
            vslc['globalframe'] = vslc['globalframe'] +
mframe
            veh_com.loc[(veh_com['videonum'] > num) &
(veh_com['Cam_ID'] == cam)] = vslc

            # If its the first video, the global frame
count starts at zero
            # else, its the max frame count of the last
video

            if i == 0:
                tot = 0

            else:
                x = i - 1
                prev = global_df['Max'][x]
                tot = tot + prev

            # Keeps a rolling list for each camera of the
added frames, so the local frame occurance of the
intersection can be easily found
            rolling = rolling.append({'Video': num,
'Total': tot, 'Cam_ID': cam}, ignore_index=True)

        cam = cam + 1

    rolling.to_csv(dirpath + '\\rolling.csv',index = False)
    print('Local and Global Frame rates calculated.')

    ped_com = ped_com.rename(columns={'frame_id':
'localframe', 'globalframe': 'frame_id'})
    veh_com = veh_com.rename(columns={'frame_id':
'localframe', 'globalframe': 'frame_id'})
```

```
    veh_trajx = veh_com.x + (veh_com.w / 2)
    veh_trajy = (veh_com.y + veh_com.h)
    veh_com['traj_x'] = veh_trajx
    veh_com['traj_y'] = veh_trajy

    ped_trajx = ped_com.x + (ped_com.w / 2)
    ped_trajy = (ped_com.y + ped_com.h)
    ped_com['traj_x'] = ped_trajx
    ped_com['traj_y'] = ped_trajy

    ped_com = ped_com
    veh_com = veh_com

    ped_com.to_csv(dirpath + '\\pedcom.csv',index = False)
    veh_com.to_csv(dirpath + '\\vehcom.csv',index = False)

else:
    rolling = pd.read_csv(dirpath + '\\rolling.csv')
    ped_com = pd.read_csv(dirpath + '\\pedcom.csv')
    veh_com = pd.read_csv(dirpath + '\\vehcom.csv')

if all_flag:
    print("All conflicts have previously been analyzed and
saved. Additional analysis will be done on the original
outputs.")
    #rolling = pd.read_csv(dirpath + '\\rolling.csv')
else:
    # In[7]:


    # #If the video does not exist, drop it from the
dataframe

    # a = ped_com.loc[(ped_com['videonum']==0)].index
    # ped_com.drop(a,inplace=True)


    # In[8]:


    ##Code above takes care of video adjustments

    # cam = 1
```

```
    # #Adjusts looks at the videos, adds the roling total
frame count to the original dataframe
    # while cam <= 4:
    #     one_rolling = rolling.loc[rolling['Cam_ID']==cam]
    #     one_rolling = one_rolling.reset_index()

    #     for i in range(len(one_rolling)):
    #         vid = one_rolling['Video'][i]
    #         add = one_rolling['Total'][i]

    #         pslc = ped_com.loc[(ped_com['Cam_ID']==cam) &
(ped_com['videonum']==vid)]
    #         vslc = veh_com.loc[(veh_com['Cam_ID']==cam) &
(veh_com['videonum']==vid)]

    #         pslc['globalframe'] = pslc['globalframe']+add
    #         vslc['globalframe'] = vslc['globalframe']+add

    #         ped_com.loc[(ped_com['Cam_ID']==cam) &
(ped_com['videonum']==vid)] = pslc
    #         veh_com.loc[(veh_com['Cam_ID']==cam) &
(veh_com['videonum']==vid)] = vslc


    #     cam = cam + 1


    # In[9]:


    # renames frame_id columns to match with previous code

    # In[10]:


    # Frames per second will depend on the type of clip you
want to run

    # Slope Check Thresholds
    #bthresh = .8
    #uthresh = 1.2

    # In[11]:
```

```
    # Adding trajectories to both the ped and veh df

    # pmaxx = ped_trajx.max()
    # pmaxy = ped_trajy.max()
    # pminx = ped_trajx.min()
    # pminy = ped_trajy.min()
    #
    # vmaxx = veh_trajx.max()
    # vmaxy = veh_trajy.max()
    # vminx = veh_trajx.min()
    # vminy = veh_trajy.min()
    #
    # minx = min(vminx,pminx) - 25
    # maxx = max(vmaxx,pmaxx) + 25
    # miny = min(vminy,pminy) - 25
    # maxy = max(vmaxy,pmaxy) + 25

    # In[12]:


    # #Adding a speed column; this cell represents vehicles
    # start_time = timeit.default_timer()

    # veh_speed = pd.DataFrame(columns =
['Cam','Index','Veh_ID','Norm_Speed'])

    # cam = 1

    # while cam <= 4:
    #     print(cam)
    #     veh_cam = veh_com.loc[veh_com['Cam_ID']==cam]
    #     veh_ids = veh_cam.object_id.unique()
    #     print('Camera '+str(cam)+' has
'+str(len(veh_ids))+' vehicles')

    #     for i in range(len(veh_ids)):

    #         if i == round(len(veh_ids))/10:
    #             print('Camera ' + str(cam) + ' is 1/10th
done')
    #         elif i == round(len(veh_ids)/5):
    #             print('Camera ' + str(cam) + ' is 1/5th
done')
    #         elif i == round(len(veh_ids)/2):
    #             print('Camera ' + str(cam) + ' is half
done')
```

```
    #             elapsed = timeit.default_timer() -
start_time
    #         elif i == round(len(veh_ids)*.75):
    #             print('Camera ' + str(cam) + ' is 3/4ths
done')

    #         veh = veh_ids[i]
    #         one_veh =
veh_cam.loc[veh_cam['object_id']==veh]
    #         indx = one_veh.index.tolist()

    #         for x in range(len(indx)):
    #             y = x + 1

    #             if y != len(indx):

    #                 pos1 = indx[x]
    #                 pos2 = indx[y]

    #                 trajx1 = one_veh['traj_x'][pos1]
    #                 trajx2 = one_veh['traj_x'][pos2]

    #                 trajy1 = one_veh['traj_y'][pos1]
    #                 trajy2 = one_veh['traj_y'][pos2]

    #                 time_dif_s =
(one_veh['frame_id'][pos2]-one_veh['frame_id'][pos1])/fps

    #                 dist =
Point((trajx1,trajy1)).distance(Point((trajx2,trajy2)))
    #                 avg_speed = dist/time_dif_s

    #                 area1 =
one_veh['h'][pos1]*one_veh['w'][pos1]
    #                 area2 =
one_veh['h'][pos2]*one_veh['w'][pos2]

    #                 avg_area =
statistics.mean([area1,area2])

    #                 norm_speed = avg_speed/avg_area

    #                 veh_speed =
veh_speed.append({'Cam':cam,'Index':pos2,'Veh_ID':veh,'Norm
_Speed':norm_speed},
```

```
    #
ignore_index = True)


    #     cam = cam + 1

    # elapsed = timeit.default_timer() - start_time


    # In[13]:


    # #####Turning Logic######
    # start_time = timeit.default_timer()

    # dot = pd.DataFrame(columns =
['Cam','Veh_Id','Dot_Product','Angle'])

    # cam = 1

    # while cam <= 4:

    #     veh_cam = veh_com.loc[veh_df['Cam_ID']==cam]
    #     veh_ids = veh_cam.object_id.unique()

    #         for i in range(len(veh_ids)):
    #         veh = veh_ids[i]
    #         one_veh =
veh_cam.loc[veh_cam['object_id']==veh].reset_index()
    #         length = len(one_veh)

    #         mid = round(length/2)


    #         #last point
    #         end = length-1


    #         a =
np.array([one_veh['traj_x'][0],one_veh['traj_y'][0]])

    #         b =
np.array([one_veh['traj_x'][mid],one_veh['traj_y'][mid]])
```

```
    #         c = b =
np.array([one_veh['traj_x'][end],one_veh['traj_y'][end]])

    #         ba = a - b
    #         bc = c - b

    #         cosine_angle = np.dot(ba, bc) /
(np.linalg.norm(ba) * np.linalg.norm(bc))
    #         angle = np.arccos(cosine_angle)
    #         angle = math.degrees(angle)

    #         dot_prod = ba @ bc

    #         dot =
dot.append({'Cam':cam,'Veh_ID':veh,'Dot_Product':dot_prod,'
Angle':angle,ignore_index = True})

    #     cam = cam + 1

    # elapsed = timeit.default_timer() - start_time


    # In[14]:


    # # TRAJECTORY ADJUSTMENT CODE. WAS INEFFECTIVE.##
    # #Will adjust toward/away traj for each camera
    # start_time = timeit.default_timer()
    # cams = veh_com.Cam_ID.unique()
    # veh_com['dir'] = 'toward'

    # # Looking at each Cam ID, so only 1 veh is adjusted
    # for i in range(len(cams)):

    #     cam_num = cams[i]

    #     veh_df = veh_com.loc[veh_com['Cam_ID']==cam_num]

    #     #Need to adjust the trajectory based on the
vehicle slope of the lines

    #     uni_vehs = veh_df.object_id.unique()

    #     #Going to loop through to see the directionality
```

```
    #     for q in range(len(uni_vehs)):

    #         #Returns the specific vehicle ID
    #         veh_ind = uni_vehs[q]

    #         adj_df = veh_com.loc[(veh_com['object_id'] ==
veh_ind) & (veh_com['Cam_ID']==cam_num)]

    #         len_adj = len(adj_df) - 1

    #         trajy1 = adj_df['traj_y'].iloc[0]
    #         trajx1 = adj_df['traj_x'].iloc[0]

    #         trajx2 = adj_df['traj_x'].iloc[-1]
    #         trajy2 = adj_df['traj_y'].iloc[-1]

    #         direct = trajy2 - trajy1

    #         if direct > 0:
    #         #this is adjusting the entire adj_df, not
just the ones in between the targeted index

    #             veh_com.loc[(veh_com['object_id'] ==
veh_ind) & (veh_com['Cam_ID']==cam_num), 'dir'] = 'away'

    #             trajy_adj = 1080 - (adj_df.y + (adj_df.h
*.75))

    #             veh_com.loc[(veh_com['object_id'] ==
veh_ind) & (veh_com['Cam_ID']==cam_num), 'traj_y'] =
trajy_adj

    #     print(i)
    # elapsed = timeit.default_timer() - start_time
    # print(elapsed)


    # In[15]:


    # List of column headers

    col_list = list(veh_com)

    # Empty Conflict Data frame outside of loop
```

```
    conflicts = pd.DataFrame(columns=col_list)
    conflicts_y = pd.DataFrame(columns=col_list)

    # Creating a new dataframe for easily read conflict
table
    cols = ['Lower FID', 'Upper FID', 'veh_object_id',
'ped_object_id', 'Category', 'Cam_ID']
    conflict_table = pd.DataFrame(columns=cols)

    # In[16]:


    # Creating a loop to parse through the data

    # start_time = timeit.default_timer()

    print('Beginning conflict identification')

    cam = 1

    while cam <= 4:

        # print(cam)

        # Need to reset frame for each camera, otherwise
the loop automatically would end
        # Time window you want to analyze
        frame = 0
        # Time window in seconds multiplied by frames
        time = 10 * fps
        # Step time in seconds multiplied by frames
        it = 1 * fps

        # Making the veh_df equal only the columns equal to
the Cam_ID
        veh_df = veh_com.loc[veh_com['Cam_ID'] == cam]
        ped_df = ped_com.loc[ped_com['Cam_ID'] == cam]

        # Max Frame ID
        vmax_frame = veh_df.frame_id.max()
        pmax_frame = ped_df.frame_id.max()

        if vmax_frame >= pmax_frame:
            max_frame = vmax_frame
```

```
        else:
            max_frame = pmax_frame

        while frame <= max_frame:

            bottom = frame
            top = frame + time

            # Section table into 3s timeframe
            ped_frame = ped_df.loc[(ped_df['frame_id'] >=
bottom) & (ped_df['frame_id'] <= top)]
            veh_frame = veh_df.loc[(veh_df['frame_id'] >=
bottom) & (veh_df['frame_id'] <= top)]

            # Gets the unique ID's for the pedestrians and
vehicles
            ped_uni = ped_frame.object_id.unique()

            ped_len = len(ped_uni)

            # to loop through unique IDs of pedestrians
            for i in range(ped_len):

                # Returns the unique ped ID ped_uni
                ped_id = ped_uni[i]
                one_ped =
ped_frame.loc[ped_frame['object_id'] == ped_id]

                # Max and Min trajectories for object
                xmax = one_ped.traj_x.max()
                xmin = one_ped.traj_x.min()

                ymax = one_ped.traj_y.max()
                ymin = one_ped.traj_y.min()

                # filter out based on whether veh exist
within the area (peds crossing horizontally)
                veh_test =
veh_frame.loc[(veh_frame['traj_x'] >= xmin) &
(veh_frame['traj_x'] <= xmax)]

                # filter out for ped vertical crossings
                veh_test_y =
veh_frame.loc[(veh_frame['traj_y'] >= ymin) &
(veh_frame['traj_y'] <= ymax)]
```

```
                # Unique vehicle IDs
                veh_uni = veh_frame.object_id.unique()

                # Identify values that occur above and
below the frame within the window (horizontal crossings)
                lower = veh_test.loc[(veh_frame['traj_y']
>= ymax)]
                upper = veh_test.loc[(veh_frame['traj_y']
<= ymin)]

                # Identify values that occur above and
below the frame within the window (vertical crossings)
                right = veh_test_y.loc[(veh_frame['traj_x']
>= xmax)]
                left = veh_test_y.loc[(veh_frame['traj_x']
<= xmin)]

                # Gets the unique object IDs between the
upper and lower values, we want to see if any match
                low_uni = lower.object_id.unique()
                up_uni = upper.object_id.unique()

                # Same, but for vertical crossings
                right_uni = right.object_id.unique()
                left_uni = left.object_id.unique()

                # Gets the length of unique ID list to
speed up the for loop for horizontal conflicts

                if low_uni.size == 0:
                    low_len = 0
                else:
                    low_len = len(low_uni)

                if up_uni.size == 0:
                    up_len = 0
                else:
                    up_len = len(up_uni)

                if up_len == 0 | low_len == 0:
                    conflicts = conflicts
                else:
                    # if statement to return the shorter
length of the two
                    if low_len > up_len:
                        small_len = up_len
```

```
                    else:
                        small_len = low_len

                    # Need to know which unique object list
to start with
                    if small_len == up_len:
                        veh_list = up_uni
                    else:
                        veh_list = low_uni

                    # Need to know which list is the list
you're looking into
                    if veh_list.tolist() ==
low_uni.tolist():
                        target = up_uni.tolist()
                    else:
                        target = low_uni.tolist()

                    # For loop to see if any unique object
ID's on the upper y bound equal the lower
                    for n in range(small_len):
                        # Returns the n object ID from the
list
                        ind = veh_list[n]

                        # see if the n position in the
shorter list matches any values of the second list
                        check = ind == target

                        # if statement to see whether the
check df is empty
                        if any(check) == False:
                            veh_con_low =
pd.DataFrame(columns=col_list)
                            veh_con_up =
pd.DataFrame(columns=col_list)
                        else:
                            veh_con_low =
lower.loc[lower['object_id'] == ind]
                            veh_con_up =
upper.loc[upper['object_id'] == ind]
                            cat = "Horizontal"
                            # making a more readable table
of conflicts
                            conflict_table =
conflict_table.append(
```

```
                                {'Lower FID': bottom,
'Upper FID': top, 'veh_object_id': ind, 'ped_object_id':
ped_id,
                                 'Category': cat, 'Cam_ID':
cam}, ignore_index=True)
                            # combining all upper and lower
values for the vehicles and appending to conflict table
                            veh_conc = [veh_con_low,
veh_con_up]
                            veh_conflicts =
pd.concat(veh_conc)

                            conflicts =
conflicts.append(veh_conflicts, ignore_index=True)
                            conflicts =
conflicts.append(one_ped, ignore_index=True)

                            # Gets the length of unique ID
list to speed up the for loop for horizontal conflicts

                if right_uni.size == 0:
                    right_len = 0
                else:
                    right_len = len(right_uni)

                if left_uni.size == 0:
                    left_len = 0
                else:
                    left_len = len(left_uni)

                if left_len == 0 | right_len == 0:
                    conflicts_y = conflicts_y
                else:
                    # if statement to return the shorter
length of the two
                    if right_len > left_len:
                        small_len_y = left_len
                    else:
                        small_len_y = right_len

                    # Need to know which unique object list
to start with
                    if small_len_y == left_len:
                        veh_list_y = left_uni
                    else:
                        veh_list_y = right_uni
```

```
                    # Need to know which list is the list
you're looking into
                    if veh_list_y.tolist() ==
right_uni.tolist():
                        target_y = left_uni.tolist()
                    else:
                        target_y = right_uni.tolist()

                    # For loop to see if any unique object
ID's on the upper bound equal the lower (vertical)
                    for z in range(small_len_y):
                        # Returns the n object ID from the
list

                        ind_y = veh_list_y[z]

                        # see if the n position in the
shorter list matches any values of the second list
                        check_y = ind_y == target_y

                        # if statement to see whether the
check df is empty
                        if any(check_y) == (str(False)):
                            veh_con_low =
pd.DataFrame(columns=col_list)
                            veh_con_up =
pd.DataFrame(columns=col_list)
                        else:
                            veh_con_low =
right.loc[right['object_id'] == ind_y]
                            veh_con_up =
left.loc[left['object_id'] == ind_y]
                            # making a more readable table
of conflicts
                            cat = 'Vertical'
                            conflict_table =
conflict_table.append(
                                {'Lower FID': bottom,
'Upper FID': top, 'veh_object_id': ind_y, 'ped_object_id':
ped_id,
                                 'Category': cat, 'Cam_ID':
cam}, ignore_index=True)

                            # combining all upper and lower
values for the vehicles and appending to conflict table
```

```
                                veh_conc = [veh_con_low,
veh_con_up]
                                veh_conflicts =
pd.concat(veh_conc)

                                conflicts =
conflicts.append(veh_conflicts, ignore_index=True)
                                conflicts =
conflicts.append(one_ped, ignore_index=True)

            frame = frame + it

        # Going to the next camera df

        cam = cam + 1
        # looptime = timeit.default_timer() - start_time

        # print(looptime)

    # elapsed = timeit.default_timer() - start_time
    # print(elapsed)


    # In[17]:


    # Drops duplicate where veh, Ped ID, and Camera Number
is the same
    conflict_drop =
conflict_table.drop_duplicates(['veh_object_id',
'ped_object_id', 'Cam_ID'])

    # Resets the conflict_drop index
    conflict_drop = conflict_drop.reset_index()

    # In[ ]:


    print('Preliminary conflict analysis is complete. There
are ' + str(len(conflict_drop)) + ' total possible
conflicts.')

    # In[ ]:
```

```
    # Tests to see the speed it takes
    # start_time = timeit.default_timer()

    # Creating false_positive and slope dataframe to see
whether or not the slope checks are accurate
    ###REMOVING FP FUNCTIONALITY, IT WAS INCONCLUSIVE###

    # fpcols = ['Cam_ID', 'Ped ID', 'Veh ID','Slope','Ped
Distance']
    # false_positive = pd.DataFrame(columns = fpcols)
    slope_df = pd.DataFrame(columns=['Slope'])

    # Empty dataframe which will be the intersection table
    int_cols = ['Cam_ID', 'PET', 'X', 'Y', 'Ped ID', 'Ped
Int Time', 'Veh ID', 'Veh Int Time', 'Type', 'Category']
    intersects = pd.DataFrame(columns=int_cols)
    sanity_cols = ['obj', 'x1', 'x2', 'y1', 'y2']
    sanity = pd.DataFrame(columns=sanity_cols)

    cam = 1

    while cam <= 4:
        # print(cam)

        # Need to reset counters for each camera
        l = 0
        x1 = 0
        x2 = 1

        # Making the veh_df equal only the columns equal to
the Cam_ID
        veh_df = veh_com.loc[veh_com['Cam_ID'] == cam]
        ped_df = ped_com.loc[ped_com['Cam_ID'] == cam]

        # This while loop goes through the conflict_drop
table to pull out the lower and upper frames, as well as
the unique object ids
        while l <= (len(conflict_drop) - 1):

            # Steps through the lower and upper frameids
from the conflict table
            lower_fid = conflict_drop['Lower FID'][l]
            upper_fid = conflict_drop['Upper FID'][l]
            veh_obj_id = conflict_drop['veh_object_id'][l]
            ped_obj_id = conflict_drop['ped_object_id'][l]
            category = conflict_drop['Category'][l]
```

```
            ped_filt = ped_df.loc[
                (ped_df['frame_id'] >= lower_fid) &
(ped_df['frame_id'] <= upper_fid) & (ped_df['object_id'] ==
ped_obj_id)]
            ped_drop = ped_filt.reset_index()

            veh_filt = veh_df.loc[
                (veh_df['frame_id'] >= lower_fid) &
(veh_df['frame_id'] <= upper_fid) & (veh_df['object_id'] ==
veh_obj_id)]
            veh_drop = veh_filt.reset_index()
            mergetest = ped_drop.merge(veh_drop,
on=['traj_x', 'traj_y'], how='inner')

            # pedx = ped_drop['traj_x']
            # pedy = ped_drop['traj_y']
            # pedcoords = []
            #
            # for i in range(len(pedx)):
            #     px = pedx[i]
            #     py = pedy[i]
            #     pedcoords.append(Point(px,py))
            #
            # vehx = ped_drop['traj_x']
            # vehy = ped_drop['traj_y']
            # vehcoords = []
            #
            # for i in range(len(vehx)):
            #     vx = vehx[i]
            #     vy = vehx[i]
            #     vehcoords.append(Point(vx,vy))
            #
            #
            # pedline = LineString(pedcoords)
            # vehline = LineString(vehcoords)
            #
            # pedveh = pedline.intersection(vehline)
            #
            # intflag = True
            #
            # try:
            #     list(pedveh.coords())
            # except:
            #     intflag = False
            #
            # if intflag:
```

```
                # Need to check if an exact match already
exists
            if mergetest.size > 0:
                xint = mergetest['traj_x'][0]
                yint = mergetest['traj_y'][0]

                # Pulls out the conflict location for veh
and peds
                ped_int = ped_df.loc[
                    (ped_df['traj_x'] == xint) &
(ped_df['traj_y'] == yint) & (ped_df['object_id'] ==
ped_obj_id)]
                veh_int = veh_df.loc[
                    (veh_df['traj_x'] == xint) &
(veh_df['traj_y'] == yint) & (veh_df['object_id'] ==
veh_obj_id)]

                # Slope Test

                ped_indx = ped_int.index
                veh_indx = veh_int.index

                ped_indx_0 = ped_indx - 1
                veh_indx_0 = veh_indx - 1
                ped_indx_1 = ped_indx + 1
                veh_indx_1 = veh_indx + 1

                ped_s =
ped_df.loc[ped_indx_0].reset_index()
                ped_e =
ped_df.loc[ped_indx_1].reset_index()
                veh_s =
veh_df.loc[veh_indx_0].reset_index()
                veh_e =
veh_df.loc[veh_indx_1].reset_index()

                px0 = ped_s['traj_x'][0]
                px1 = ped_e['traj_x'][0]
                py0 = ped_s['traj_y'][0]
                py1 = ped_e['traj_y'][0]

                vx0 = veh_s['traj_x'][0]
                vx1 = veh_e['traj_x'][0]
                vy0 = veh_s['traj_y'][0]
                vy1 = veh_e['traj_y'][0]
```

```
                ped_slope = (py1 - py0) / (px1 - px0)
                veh_slope = (vy1 - vy0) / (vx1 - vx0)

                #slope = ped_slope / veh_slope
                #slope_df = slope_df.append({'Slope':
slope}, ignore_index=True)

                # if (slope <= uthresh) and (slope >=
bthresh):
                # print('Possible FP')
                # false_positive =
false_positive.append({'Cam_ID':cam, 'Ped ID':ped_obj_id,
'Veh ID':veh_obj_id,'Slope':slope,'Ped
Distance':ped_distance}, ignore_index = True)

                ped_int = ped_int.reset_index()
                veh_int = veh_int.reset_index()

                # Calculates the absolute value of frame
differential, eventually will include directionality
                # to get the type of collision and if it
was a turning movement

                veh_s = (veh_int.frame_id[0]) / fps
                ped_s = (ped_int.frame_id[0]) / fps
                PET = ((veh_int.frame_id[0]) -
(ped_int.frame_id[0])) / fps
                int_type = 'TBD'

                # Creating a singular intersection table
with the specific IDs for one intersection
                int_table = pd.DataFrame([[cam, PET, xint,
yint, ped_obj_id, ped_s, veh_obj_id, veh_s, int_type,
category]],
                                         columns=int_cols)

                # Appends the int_table to the intersection
table
                intersects = intersects.append(int_table,
ignore_index=True)

                # need to skip the next while loop
                ped_drop = []

            # Give the pedestrian line for a given conflict
```

```
            while x1 <= (len(ped_drop) - 2):

                # Need to reset the veh_line segment loop
                veh_x1 = 0
                veh_x2 = 1

                xo = ped_drop['traj_x'][x1]
                yo = ped_drop['traj_y'][x1]

                x_1 = ped_drop['traj_x'][x2]
                y_1 = ped_drop['traj_y'][x2]
                ped_line_segment = LineString([(xo, yo),
(x_1, y_1)])

                # Now we need to see if this specific
pedestrian line segment intersects with the entire veh line
                while veh_x1 <= (len(veh_drop) - 2):

                    # Default int_seg_check_value
                    int_seg_check = 1
                    veh_xo = veh_drop['traj_x'][veh_x1]
                    veh_x_1 = veh_drop['traj_x'][veh_x2]
                    veh_yo = veh_drop['traj_y'][veh_x1]
                    veh_y1 = veh_drop['traj_y'][veh_x2]
                    veh_line_segment = LineString([(veh_xo,
veh_yo), (veh_x_1, veh_y1)])

                    # Test for an intersection for this
line segment
                    int_test =
ped_line_segment.intersection(veh_line_segment)

                    try:
                        list(int_test.coords)
                    except Exception:
                        int_seg_check = 0

                    if int_seg_check == 1:
                        # Getting the two coordinates of
the pedestrian if there is a intersection
                        int_test =
ped_line_segment.intersection(veh_line_segment)
                        ped_first_x = xo
                        ped_first_y = yo
                        ped_second_x = x_1
                        ped_second_y = y_1
```

```
                        # san_obj = 'Ped'
                        # ped_sanity =
pd.DataFrame([[san_obj,ped_first_x, ped_first_y,
ped_second_x, ped_second_y]], columns=sanity_cols)
                        # sanity =
sanity.append(ped_sanity, ignore_index = True)
                        first_frame =
ped_df.loc[(ped_df['traj_x'] == ped_first_x) &
(ped_df['traj_y'] == ped_first_y) & (
                                        ped_df['object_id'] ==
ped_obj_id)]
                        second_frame = ped_df.loc[
                            (ped_df['traj_x'] ==
ped_second_x) & (ped_df['traj_y'] == ped_second_y) & (
                                           ped_df['object_id']
== ped_obj_id)]
                        first_frame =
first_frame['frame_id']
                        first_frame = list(first_frame)
                        first_frame = first_frame[0]
                        second_frame =
second_frame['frame_id']
                        second_frame = list(second_frame)
                        second_frame = second_frame[0]
                        ped_distance = Point((xo,
yo)).distance(Point((x_1, y_1)))
                        frame_diff = second_frame -
first_frame
                        frame_per_distance = frame_diff /
ped_distance

                        # If for some reason two
coordinates overlap on the linestring
                        if len(int_test.coords) > 1:
                            # print('This may be the
error')
                            int_test = int_test.coords[0]
                            int_test = Point(int_test)

                        interp_distance = Point((xo,
yo)).distance(Point(int_test))
                        ped_interp_frame = first_frame +
frame_per_distance * interp_distance

                        # Now do the same thing for the
vehicle
                        veh_first_x = veh_xo
```

```
                        veh_first_y = veh_yo
                        veh_second_x = veh_x_1
                        veh_second_y = veh_y1

                        ####Slope Test#####

                        #ped_slope = (ped_second_y -
ped_first_y) / (ped_second_x - ped_first_x)
                        #veh_slope = (veh_second_y -
veh_first_y) / (veh_second_x - veh_first_x)

                        ####Now need to test the slopes and
see how much they align by, a threshold point###
                        #slope = ped_slope / veh_slope
                        #slope_df =
slope_df.append({'Slope': slope}, ignore_index=True)

                        # if (slope <= uthresh) and (slope
>= bthresh):
                        # print('Possible FP')
                        # false_positive =
false_positive.append({'Cam_ID':cam, 'Ped ID':ped_obj_id,
'Veh ID':veh_obj_id,'Slope':slope, 'Ped
Distance':ped_distance}, ignore_index = True)

                        # san_obj = 'Veh'
                        # veh_sanity =
pd.DataFrame([[san_obj, veh_first_x, veh_first_y,
veh_second_x, veh_second_y]], columns=sanity_cols)
                        # sanity =
sanity.append(veh_sanity, ignore_index = True)
                        vfirst_frame =
veh_df.loc[(veh_df['traj_x'] == veh_first_x) &
(veh_df['traj_y'] == veh_first_y) & (
                                        veh_df['object_id'] ==
veh_obj_id)]
                        vsecond_frame = veh_df.loc[
                            (veh_df['traj_x'] ==
veh_second_x) & (veh_df['traj_y'] == veh_second_y) & (
                                            veh_df['object_id']
== veh_obj_id)]
                        vfirst_frame =
vfirst_frame['frame_id']
                        vfirst_frame = list(vfirst_frame)
                        vfirst_frame = vfirst_frame[0]
                        vsecond_frame =
vsecond_frame['frame_id']
```

```
                        vsecond_frame = list(vsecond_frame)
                        vsecond_frame = vsecond_frame[0]
                        veh_distance = Point((veh_xo,
veh_yo)).distance(Point(veh_x_1, veh_y1))
                        vframe_diff = vsecond_frame -
vfirst_frame
                        vframe_per_distance = vframe_diff /
veh_distance
                        vinterp_distance = Point((veh_xo,
veh_yo)).distance(Point(int_test))
                        veh_interp_frame = vfirst_frame +
vframe_per_distance * vinterp_distance

                        # Exit nested loops
                        veh_x1 = len(veh_drop)
                        x1 = len(ped_drop)
                        intx = int_test.x
                        inty = int_test.y
                        ped_s = ped_interp_frame / fps
                        veh_s = veh_interp_frame / fps
                        PET = veh_interp_frame -
ped_interp_frame
                        PET = PET / fps
                        Type = 'tbd'
                        int_table = pd.DataFrame(
                            [[cam, PET, intx, inty,
ped_obj_id, ped_s, veh_obj_id, veh_s, Type, category]],
                            columns=int_cols)
                        intersects =
intersects.append(int_table, ignore_index=True)

                    # Outside of if statement to keep
looping through
                    veh_x1 = veh_x1 + 1
                    veh_x2 = veh_x2 + 1

                # Keep looping through the pedline if
there's not interesections within the entire veh line
                x1 = x1 + 1
                x2 = x2 + 1

            # Goes to the next instance of the conflict
table
            l = l + 1

            # Reset the loops for the next instance on the
conflict table
```

```
            x1 = 0
            x2 = 1
            veh_x1 = 0
            veh_x2 = 1

        cam = cam + 1

    # Calculate time elapsed for this block
    # elapsed = timeit.default_timer() - start_time
    # print(elapsed)

    print('PET calculations are complete. Now generating
playback functionality')
    # ' There are ' +str(len(false_positive)) + ' possible
false positives.')


    # In[30]:


    ctest = conflict_drop.sort_values(by=['ped_object_id',
'veh_object_id'])
    inttest = intersects.sort_values(by=['Ped ID', 'Veh
ID'])
    ctest = ctest.reset_index()
    pedframe = intersects['Ped Int Time'] * fps
    intersects['Ped Frame'] = pedframe
    intersects = intersects.sort_values(by=['Ped Frame'])
    intersects = intersects.sort_values(by=['Cam_ID', 'Ped
Frame'])

    # In[31]:


    intersects = intersects.rename(columns={'Ped Frame':
'Frame'})

    # In[85]:


    # Gets the larger frame_time to look at

    for i in range(len(intersects)):
        pedt = intersects['Ped Int Time'][i]
        veht = intersects['Veh Int Time'][i]
```

```
        if pedt <= veht:
            frm = pedt * fps
        else:
            frm = veht * fps

        intersects['Frame'][i] = frm

# In[33]:
placeholder = 1

while placeholder <= 2:
    # Shrinking down conflicts to the conflicts we really
care about

    if all_flag:
        placeholder = 2
        intersects = allconflicts

    if placeholder == 1:
        path = dirpath
        cam1 = path + "\\" + "Camera1"
        cam2 = path + "\\" + "Camera2"
        cam3 = path + "\\" + "Camera3"
        cam4 = path + "\\" + "Camera4"
        cutoff = 10
        indexnames =
intersects.loc[(intersects['PET']>cutoff) |
(intersects['PET']<(cutoff*-1))].index
        intersects.drop(indexnames, inplace=True)
        intersects = intersects.reset_index(drop=True)
        print('All conflicts are being analyzed and written
to the folder.')
    else:
        path = secondpath
        cam1 = path + "\\" + "Camera1"
        cam2 = path + "\\" + "Camera2"
        cam3 = path + "\\" + "Camera3"
        cam4 = path + "\\" + "Camera4"
        if behind_flag:
            indexnames =
intersects.loc[(intersects['PET']>int(ccut)) |
(intersects['PET']<(int(bc)*-1))].index
            intersects.drop(indexnames, inplace=True)
            intersects = intersects.reset_index(drop=True)
        else:
            indexnames = intersects.loc[(intersects['PET']
> int(ccut)) | (intersects['PET'] < 0)].index
```

```
            intersects.drop(indexnames, inplace=True)
            intersects = intersects.reset_index(drop=True)
    print("Targeted conflicts are being analyzed and
written to the folder.")

    # In[86]:


    play = intersects.copy()
    play['localframe']=0
    play['vid']=0
    cam = 1

    while cam <= 4:
        tar = rolling.loc[rolling['Cam_ID']==cam]
        tar = tar.sort_values(by = 'Total',
ascending=False)
        tar = tar.reset_index()

        for i in range(len(tar)):
            frm = tar['Total'][i]
            vid = tar['Video'][i]
            slc = play.loc[(play['Cam_ID']==cam) &
(play['Frame']>=frm) & (play['localframe']==0)]

            #Calculating back the local frame rate for
playback detect capabilities
            slc['localframe']=slc['Frame']-frm

            slc['vid']=vid

            play.loc[(play['Cam_ID']==cam) &
(play['Frame']>=frm) & (play['localframe']==0)] = slc

        cam = cam + 1


    # In[87]:


    cam = 1
    play['filename']=''

    while cam <= 4:
        df = ped_com.loc[ped_com['Cam_ID']==cam]
```

```
        videos = df.videonum.unique()
        filenames = df.filename.unique()

        #If the number of videos is not the number of
files, something went wrong

        if len(videos)!=len(filenames):
            print('Something is wrong')
            break

        #Matching the filenames to the order it occurs
        for i in range(len(videos)):

            vid = videos[i]
            filename = filenames[i]

            slc = play.loc[(play['Cam_ID']==cam) &
(play['vid']==vid)]
            slc['filename']=filename

            play.loc[(play['Cam_ID']==cam) &
(play['vid']==vid)]=slc

        cam = cam + 1


    # In[36]:


    # Writing Conflict files for each camera to csv
    cam = 1
    while cam <= 4:

        out = intersects.loc[intersects['Cam_ID']==cam]

        if cam==1:
            string = cam1 + '\\' + 'Conflicts.csv'
        elif cam ==2:
            string = cam2 + '\\' + 'Conflicts.csv'
        elif cam ==3:
            string = cam3 + '\\' + 'Conflicts.csv'
        elif cam ==4:
            string = cam4 + '\\' + 'Conflicts.csv'

        out.to_csv(string, index=False)
```

```
        cam = cam + 1


    # In[37]:


    play_df =
play[['Cam_ID','X','Y','filename','localframe','Ped
ID']].copy()
    play_df['localframe']=play_df['localframe']/fps
    play_df = play_df.round({'localframe':1})
    play_df['time'] = 0
    play_df['slope'] = 0
    play_df['cam_id'] = siteID

    #Reordering column numbers and renaming to match
existing structure
    play_df = play_df.rename(columns =
{'X':'x_start','Y':'y_start','Video':'filename','Ped ID':
'pedID', 'localframe':'frame_time'})
    play_df.index.name = 'ser_num'
    play_df =
play_df[['Cam_ID','time','slope','pedID','x_start','y_start
','filename','frame_time','cam_id']]
    play_df = play_df.round({'x_start': 0,'y_start': 0})
    play_df = play_df.round({'frame_time':0})


    # In[38]:


    ###Seeing if sycning time w/ pedID will load into
playback detect tool correctly

    play_df['time']=play_df['pedID']

    cam = 1

    while cam <= 4:
        df = play_df.loc[play_df['Cam_ID']==cam]
        df = df.reset_index(drop = True)
        df = df.drop(columns = 'Cam_ID')
        df.index.name = 'ser_num'
```

```
        if cam==1:
            string = cam1 + '\\' +
'camera_1_ped_coordinate.txt'
        elif cam ==2:
            string = cam2 + '\\' +
'camera_2_ped_coordinate.txt'
        elif cam ==3:
            string = cam3 + '\\' +
'camera_3_ped_coordinate.txt'
        elif cam ==4:
            string = cam4 + '\\' +
'camera_4_ped_coordinate.txt'

        line =
'============================================================
======\n'
        df.index = df.index + 1

        string_df = df.to_csv(header=None, index=True)



        f = open(string, 'w+')

        i = 1

        while i <= 6:
            f.write(line)

            if i == 5:
                f.write(string_df)

            i = i + 1

        cam = cam + 1
        f.close()
    print('Playback detect compatibility complete.')



        #df.index = df.index + 1
        #df.to_csv(string, index=True)

        #cam = cam + 1
```

```
    #print('Playback detect compatability complete.')


    # In[39]:


    ##This portion of code makes validation easier

    validation = intersects.drop(columns = ['X','Y','Ped
Int Time','Veh Int Time','Type','Category','Frame'])
    validation['time']=play_df['time']
    cam = 1
    while cam <= 4:
        val = validation.loc[validation['Cam_ID']==cam]
        if cam==1:
            string2 = cam1 + '\\' + 'validation.csv'
        elif cam ==2:
            string2 = cam2 + '\\' + 'validation.csv'
        elif cam ==3:
            string2 = cam3 + '\\' + 'validation.csv'
        elif cam ==4:
            string2 = cam4 + '\\' + 'validation.csv'

        val.to_csv(string2, index=False)
        cam = cam + 1


    # In[40]:


    ######Going to want to save these plots in a directory
for the Site ID and Camera######
    print('Now saving images')

    xminlist = list()
    xmaxlist = list()
    yminlist = list()
    ymaxlist = list()

    cam = 1

    while cam <= 4:
        #Resetting following loop
        s = 0
        #closing figure
        plt.close()
```

```
        veh_df = veh_com.loc[veh_com['Cam_ID']==cam]
        ped_df = ped_com.loc[ped_com['Cam_ID']==cam]

        adj_intersects =
intersects.loc[(intersects['Cam_ID']==cam)]
        adj_intersects = adj_intersects.reset_index()

        #Need to plot instances of the conflicts

        while s < len(adj_intersects):
            #Gets the unique ID for peds and vehicles
            ped_ind = adj_intersects['Ped ID'][s]
            veh_ind = adj_intersects['Veh ID'][s]

            #Gets the time frame for to plot. Do not want
to plot entire trajectory without
            #Time implement because it would get incredibly
messy
            lower_plot = (adj_intersects['Ped Int
Time'][s])*fps - time
            upper_plot = (adj_intersects['Ped Int
Time'][s])*fps + time

            ped_plot = ped_df[(ped_df['frame_id'] >=
lower_plot) &
                              (ped_df['frame_id'] <=
upper_plot) & (ped_df['object_id']==ped_ind)]

            veh_plot = veh_df[(veh_df['frame_id'] >=
lower_plot) &
                              (veh_df['frame_id'] <=
upper_plot) & (veh_df['object_id']==veh_ind)]

            ped_plot_test = ped_plot[['traj_x','traj_y']]
            veh_plot_test = veh_plot[['traj_x','traj_y']]

            pminx = ped_plot_test['traj_x'].min()
            pmaxx = ped_plot_test['traj_x'].max()
            pminy = ped_plot_test['traj_y'].min()
            pmaxy = ped_plot_test['traj_y'].max()

            xminlist.append(pminx)
            xmaxlist.append(pmaxx)
            yminlist.append(pminy)
            ymaxlist.append(pmaxy)
```

```
            vminx = veh_plot_test['traj_x'].min()
            vmaxx = veh_plot_test['traj_x'].max()
            vminy = veh_plot_test['traj_y'].min()
            vmaxy = veh_plot_test['traj_y'].max()

            xminlist.append(vminx)
            xmaxlist.append(vmaxx)
            yminlist.append(vminy)
            ymaxlist.append(vmaxy)

            minx = min(xminlist) - 25
            maxx = max(xmaxlist) + 25
            miny = min(yminlist) - 25
            maxy = max(ymaxlist) + 25

            pet_check = adj_intersects['PET'][s]

            int_point_x = adj_intersects['X'][s]
            int_point_y = adj_intersects['Y'][s]


            ped_lines =
plt.plot(ped_plot_test['traj_x'],ped_plot_test['traj_y'],'B
lue')
            veh_lines =
plt.plot(veh_plot_test['traj_x'],veh_plot_test['traj_y'],'O
range')


            if pet_check < 0:
                if pet_check >= -1:
                    int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
10, color="black")
                elif pet_check >= -3:
                    int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
7, color="black")
                else:
                    int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
5, color="black")

            if pet_check >= 0:
                if pet_check <= 1:
```

```
                    int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
10, color="red")
                elif pet_check <= 3:
                    int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
7, color="red")
                else:
                    int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
5, color="red")


            if cam==1:
                string = cam1 + '\\' + 'Conflict_Plot.png'
            elif cam ==2:
                string = cam2 + '\\' + 'Conflict_Plot.png'
            elif cam ==3:
                string = cam3 + '\\' + 'Conflict_Plot.png'
            elif cam ==4:
                string = cam4 + '\\' + 'Conflict_Plot.png'



            ax = plt.gca()
            ax.set(xlim=(minx, maxx), ylim=(miny, maxy))
            plt.gca().invert_yaxis()
            plt.savefig(string)

            #Saving figure in proper directory

            #Adding texts to the intersects
            #plt.text(int_point_x+5, int_point_y+20, s,
fontsize = 12)

            s=s+1

        cam = cam + 1

    plt.close()



    # In[50]:
```

```
    #Plotting one ped trajectory to assist with validation
    camID = 1


    while camID <=4:

        veh_df = veh_com.loc[veh_com['Cam_ID']==camID]
        ped_list =
intersects.loc[intersects['Cam_ID']==camID]
        ped_uni = ped_list['Ped ID'].unique()

        z = 0

        while z < len(ped_uni):
            pedID = ped_uni[z]
            ped_cam = ped_com.loc[ped_com['Cam_ID']==camID]
            ped_one =
ped_cam.loc[ped_cam['object_id']==pedID]
            # print(ped_one.head())
            plot = ped_one[['traj_x','traj_y']]


            line = plt.plot(plot['traj_x'],plot['traj_y'])

            veh_list = intersects.loc[(intersects['Ped
ID']==pedID) & (intersects['Cam_ID']==camID)]

            z = z + 1
            y = 0

            while y < len(veh_list):

                veh_uni = veh_list['Veh ID'].unique()
                vehID = veh_uni[y]
                veh =
veh_com.loc[(veh_com['Cam_ID']==camID) &
(veh_com['object_id']==vehID)]

                one_int = intersects.loc[(intersects['Ped
ID']==pedID) & (intersects['Veh ID']==vehID)
                                                    &
(intersects['Cam_ID']==camID)]
```

```
                one_int = one_int.reset_index(drop=True)

                pet_check = one_int['PET'][0]
                int_point_x = one_int['X'][0]
                int_point_y = one_int['Y'][0]
                veh_time = one_int['Veh Int Time'][0]

                lower_plot = (veh_time - 10)*fps
                upper_plot = (veh_time + 10)*fps


                veh_plot = veh_df[(veh_df['frame_id'] >=
lower_plot) &
                                  (veh_df['frame_id'] <=
upper_plot) & (veh_df['object_id']==vehID)]

                veh_plot_test =
veh_plot[['traj_x','traj_y']]



                veh_lines =
plt.plot(veh_plot_test['traj_x'],veh_plot_test['traj_y'],'O
range')

                if pet_check < 0:
                    if pet_check >= -1:
                        int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
10, color="black")
                    elif pet_check >= -3:
                        int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
7, color="black")
                    else:
                        int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
5, color="black")

                if pet_check >= 0:
                    if pet_check <= 1:
                        int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
10, color="red")
                    elif pet_check <= 3:
```

```
                        int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
7, color="red")
                    else:
                        int_point_plot =
plt.plot(int_point_x,int_point_y, marker = 'o',markersize =
5, color="red")

                y = y+1

            if camID ==1:
                string = cam1 + '\\' + 'Ped_' + str(pedID)
+ 'conflicts.png'
            elif camID ==2:
                string = cam2 + '\\' + 'Ped_' + str(pedID)
+ 'conflicts.png'
            elif camID ==3:
                string = cam3 + '\\' + 'Ped_' + str(pedID)
+ 'conflicts.png'
            elif camID ==4:
                string = cam4 + '\\' + 'Ped_' + str(pedID)
+ 'conflicts.png'

            ax = plt.gca()
            ax.set(xlim=(minx,maxx),ylim=(miny,maxy))
            plt.gca().invert_yaxis()
            title = 'Ped ' + str(pedID) + ' Conflict Plot'
            plt.title(title, fontsize=20)
            plt.savefig(string)
            plt.close()

        camID = camID + 1


    # In[46]:


    ##### Intersection Statistics for Each camera

    cam = 1

    #####Need to define different cutoff points for
conflicts#####
    ####Behind Conflict, Behind Near Miss, Behind Severe

    ####Severe Conflict Defined as PET <=1, >0
```

```
    col_list = ["Camera","Ped Count","Total
Conflicts","Behind Conflict", "Behind Near Miss", "Behind
Severe","Conflict","Near Miss","Severe"]
    cols = ['Camera','Conflict/Ped %','Behind %','Conflict
%','Near Miss %', 'Severe %']
    per = pd.DataFrame(columns = cols)
    stats = pd.DataFrame(columns = col_list)
    cutoff = 0
    bnearmiss = -3
    bsevere = -1
    conf = 10
    nearmiss = 3
    severe = 1

    while cam<=4:
        i = cam - 1
        int_df = intersects.loc[intersects['Cam_ID']==cam]
        total = len(int_df)
        behind_con = len(int_df.loc[int_df['PET']<cutoff])
        bnear_con = len(int_df.loc[(int_df['PET']<cutoff) &
(int_df['PET']>=bnearmiss)])
        bsev_con = len(int_df.loc[(int_df['PET']<cutoff) &
(int_df['PET']>=bsevere)])
        conf_con = len(int_df.loc[int_df['PET']>=cutoff])
        near_con = len(int_df.loc[(int_df['PET']>=cutoff) &
(int_df['PET']>=nearmiss)])
        sev_con = len(int_df.loc[(int_df['PET']>=cutoff) &
(int_df['PET']<=severe)])

        if total == 0:
            beh_per = "NA"
            con_per = "NA"
            near_per = "NA"
            sev_per = "NA"

        else:
            beh_per = round(behind_con/total*100,1)
            con_per = round(conf_con/total*100,1)
            near_per = round(near_con/total*100,1)
            sev_per = round(sev_con/total*100,1)


        objects = ["Total","Behind", "B Near", "B
Sev","Conflict","Near","Sev"]
        x_pos = [i for i, _ in enumerate(objects)]
```

```
        y_val =
[total,behind_con,bnear_con,bsev_con,conf_con,near_con,sev_
con]

        plt.bar(x_pos, y_val, align='center', alpha=0.5,
width = 0.3)
        plt.xticks(x_pos,objects)
        plt.ylabel('Number of Conflicts')
        plt.title('Conflicts by Type and Occurance')


        if cam ==1:
            string = cam1 + '\\' + 'Conflict_Stats.png'
        elif cam ==2:
            string = cam2 + '\\' + 'Conflict_Stats.png'
        elif cam ==3:
            string = cam3 + '\\' + 'Conflict_Stats.png'
        elif cam ==4:
            string = cam4 + '\\' + 'Conflict_Stats.png'

        plt.savefig(string)
        plt.close()

        slc = ped_com.loc[ped_com['Cam_ID']==cam]
        ped_count = len(slc.object_id.unique())
        if ped_count == 0:
            conflict_ped = 'NA'
        else:
            conflict_ped = round(total/ped_count*100)



        per = per.append({'Camera' : cam, 'Conflict/Ped %'
: conflict_ped, 'Behind %' : beh_per, 'Conflict %' :
con_per, 'Near Miss %' : near_per, 'Severe %' : sev_per},
ignore_index=True)
        stats = stats.append({'Camera' : cam, 'Ped Count' :
ped_count, 'Total Conflicts' : total,  'Behind Conflict' :
behind_con,
                              'Behind Near Miss' :
bnear_con, 'Behind Severe' : bsev_con, 'Conflict' :
conf_con,
                              'Near Miss' : near_con,
'Severe' : sev_con}, ignore_index=True)
```

```
        cam = cam+1

    string = path + "\\" + "Conflict_Stats.txt"
    string2 = path + "\\" + "Conflict_Per.txt"
    #string3 = dirpath + "\\" + "False_Pos.txt"

    stats.to_csv(string, index=False)
    per.to_csv(string2, index=False)
    #false_positive.to_csv(string3, index=False)
    placeholder = placeholder + 1
    # In[43]:


print("The Conflict Analysis has been completed. Please
check the Report file directory.")
elapsed = timeit.default_timer() - start_time

elapsed = round(elapsed/60,1)

if elapsed >= 60:
    elapsed = round(elapsed/60,1)
    print('Total processing time is ' + str(elapsed) + '
hours.')

else:
    print('Total processing time is ' + str(elapsed) + '
minutes.')
```

REFERENCES

Cherry, Christopher, Brian Donlon, Xuedong Yan, Samuel Elliott Moore, and Jian Xiong. “Illegal Mid-Block Pedestrian Crossings in China: Gap Acceptance, Conflict and Crossing Path Analysis.” International Journal of Injury Control and Safety Promotion19, no. 4 (2012): 320–30. https://doi.org/10.1080/17457300.2011.628751.

Cui, Zhenzhong, and Shashi S. Nambisan. “Methodology for Evaluating the Safety of Midblock Pedestrian Crossings.” Transportation Research Record 1828, no. 1 (January 2003): 75–82. doi:10.3141/1828-09.

Ishaque, Muhammad Moazzam, and Robert B. Noland. “Behavioural Issues in Pedestrian Speed Choice and Street Crossing Behaviour: A Review.” Transport Reviews28, no. 1 (2008): 61–85. https://doi.org/10.1080/01441640701365239.

Ismail, Karim, Tarek Sayed, Nicolas Saunier, and Clark Lim. “Automated Analysis of Pedestrian–Vehicle Conflicts Using Video Data.” Transportation Research Record 2140, no. 1 (January 2009): 44–54. doi:10.3141/2140-05.

Ismail, Karim, Tarek Sayed, and Nicolas Saunier. “Methodologies for Aggregating Indicators of Traffic Conflict.” Transportation Research Record: Journal of the Transportation Research Board2237, no. 1 (2011): 10–19. https://doi.org/10.3141/2237-02.

Ismail, Karim, Tarek Sayed, and Nicolas Saunier. “A Methodology for Precise Camera Calibration for Data Collection Applications in Urban Traffic Scenes.” Canadian Journal of Civil Engineering40, no. 1 (2013): 57–67. https://doi.org/10.1139/cjce-2011-0456.

Jiang, Xiaobei, Wuhong Wang, Klaus Bengler, and Weiwei Guo. “Analyses of Pedestrian Behavior on Mid-Block Unsignalized Crosswalk Comparing Chinese and German Cases.” Advances in Mechanical Engineering, (November 2015). doi:10.1177/1687814015610468.

Laureshyn, A., & Ardö, H. (2006). Automated video analysis as a tool for road user behavior. In R. Risser (Ed.), [Host publication title missing] Proceedings of ITS World Congress, London, 8-12 October 2006.

Malkhamah, Siti, Miles Tight, and Frank Montgomery. “The Development of an Automatic Method of Safety Monitoring at Pelican Crossings.” Accident Analysis & Prevention37, no. 5 (2005): 938–46. https://doi.org/10.1016/j.aap.2005.04.012.

Matsui, Yasuhiro, Masahito Hitosugi, Tsutomu Doi, Shoko Oikawa, Kunio Takahashi, and Kenichi Ando. “Features of Pedestrian Behavior in Car-to-Pedestrian Contact Situations in Near-Miss Incidents in Japan.” Traffic Injury Prevention14, no. sup1 (2013). https://doi.org/10.1080/15389588.2013.796372.

Saunier, Nicolas, and Tarek Sayed. “Automated Analysis of Road Safety with Video Data.” Transportation Research Record: Journal of the Transportation Research Board2019, no. 1 (2007): 57–64. https://doi.org/10.3141/2019-08.

Saunier, Nicolas, and Tarek Sayed. “Probabilistic Framework for Automated Analysis of Exposure to Road Collisions.” Transportation Research Record 2083, no. 1 (January 2008): 96–104. doi:10.3141/2083-11.

Sayed, Tarek, Mohamed H. Zaki, and Jarvis Autey. “Automated Safety Diagnosis of Vehicle–Bicycle Interactions Using Computer Vision Analysis.” Safety Science59 (2013): 163–72. https://doi.org/10.1016/j.ssci.2013.05.009.

Schneider, Robert J, Rhonda M Ryznar, and Asad J Khattak. “An Accident Waiting to Happen: a Spatial Approach to Proactive Pedestrian Planning.” Accident Analysis & Prevention36, no. 2 (2004): 193–211. https://doi.org/10.1016/s0001-4575(02)00149-5.

St-Aubin, Paul, Nicolas Saunier, and Luis Miranda-Moreno. “Large-Scale Automated Proactive Road Safety Analysis Using Video Data.” Transportation Research Part C: Emerging Technologies58 (2015): 363–79. https://doi.org/10.1016/j.trc.2015.04.007.

Svensson, Åse, and Christer Hydén. “Estimating the Severity of Safety Related Behaviour.” Accident Analysis & Prevention38, no. 2 (2006): 379–85. https://doi.org/10.1016/j.aap.2005.10.009.

“Traffic Safety Facts: 2017 Data Pedestrians.” (2019). NHTSA. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812681

Usher, Colin, and Wayne Daley. “Extended Field Testing and Enhancement of a Portable Pedestrian and Cyclist Detection System”. GDOT. (2020).

Watkins, Kari, Michael Rodgers, Randall Guensler, and Yanzhi Xu. “BICYCLE AND PEDESTRIAN SAFETY IN THE HIGHWAY SAFETY MANUAL.” GDOT. (2016).

Wu, Jiawei, Essam Radwan and Hatem Abou-Senna. “Pedestrian-vehicle conflict analysis at signalized intersections using micro-simulation.” (2016).